Amtiri

REFERENCE MANUAL

Modbus

Generated on November 25, 2026

Modbus is the plainest way to exchange values with a field device: numbered registers and bits, read and written with a handful of function codes. Data Orchester speaks it in both directions and over both transports. A **master** polls a device and writes to it; a **slave** answers a remote master from variables of its own. This page covers the four module types: their settings, how registers become values, and what each one reports.

Module	What it does	Typical use
Modbus TCP Master (`MB-TCP-Master`)	Reads a device over TCP into variables on a schedule, and writes coils and holding registers when variables change	PLCs, meters, drives and gateways on Ethernet
Modbus Serial Master (`MB-SER-Master`)	The same over Modbus RTU on a serial line, with every device of the line in one module	RS-485 meters, drives and controllers
Modbus TCP Slave (`MB-TCP-Slave`)	Answers a remote master from four list variables; what the master writes lands in them	A SCADA or HMI that reads and writes Data Orchester
Modbus Serial Slave (`MB-SER-Slave`)	The same over Modbus RTU on a serial port	A PLC or RTU that polls over RS-485

Every type also has the settings all modules share: whether it is enabled, its name, its status and message variables and their thresholds. They are described once, in Settings every module shares.

Registers, addresses and values

Both masters describe each row with a **register type** (`remoteType`), which chooses the function code and what the variable receives, and an **address** (`remoteAddress`).

Register types

A collector reads:

Register type	Editor label	Function code	Reads	The variable gets
`bool`	FC01 - Coil	01	1 coil	`boolean`
`input_bool`	FC02 - Discrete input	02	1 discrete input	`boolean`
`int`	FC03 - Holding register (int16)	03	1 holding register	`int`, 0 to 65535
`input_int`	FC04 - Input register (int16)	04	1 input register	`int`, 0 to 65535
`float`	FC03 - Holding registers (float32)	03	2 holding registers	`float`
`input_float`	FC04 - Input registers (float32)	04	2 input registers	`float`
`array_int`	FC03 - Holding registers (int16 array)	03	Quantity holding registers	`[int]`, a list
`array_input_int`	FC04 - Input registers (int16 array)	04	Quantity input registers	`[int]`, a list
`array_bool`	FC01 - Coils (array)	01	Quantity coils	`[boolean]`, a list
`array_input_bool`	FC02 - Discrete inputs (array)	02	Quantity discrete inputs	`[boolean]`, a list

A forwarder writes:

Register type	Editor label	Function code	Writes	The variable holds
`bool`	FC05 - Coil	05	1 coil	a `boolean`
`int`	FC06 - Holding register (int16)	06	1 holding register	a number

When the editor's variable picker creates a row's variable, it gives it the type in the last column. A variable declared with a type the register cannot fill, such as a `float` read into an `int`, is named in the engine log when the configuration loads, and a value that does not fit the variable is refused rather than stored.

Addresses

The address is the offset carried in the Modbus request, counted from 0, and it is sent exactly as written. The register type already chooses the table, so the table digit of the numbering many device manuals use is not part of it:

The manual says	Table	Address to enter
00001	Coil	0
10001	Discrete input	0
30001	Input register	0
40001	Holding register	0
40010 (or 400010)	Holding register	9

A manual that already counts from 0 gives the address as it is. The editor names the address **Start** in a collector row and **Register** in a forwarder row.

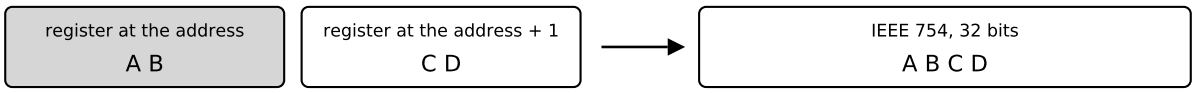
> **Tip**
>
> When a manual is unclear, read one register whose value you know, such as a setpoint shown on the device's display. A value that turns up one address away tells you which way the manual counts.

Values and word order

Addresses count from 0

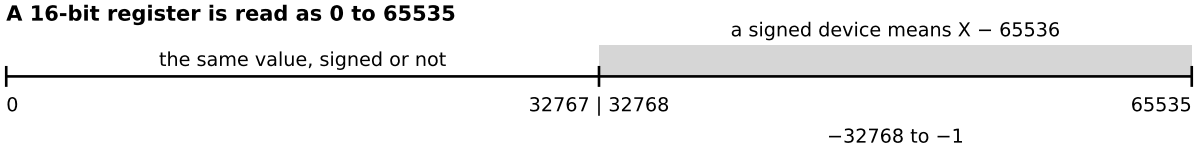


A float spans two registers, high word first



A device that sends the low word first needs the two registers swapped in a formula.

A 16-bit register is read as 0 to 65535



- **A 16-bit register is unsigned.** The editor labels say int16, but `int` and `input_int` give 0 to 65535. When the device means a signed value, read the register into one variable and give a second variable the formula `if(DT_205_RAW > 32767, DT_205_RAW - 65536, DT_205_RAW)`. A scaled value divides in the same formula.
- **A float is two registers, high word first.** `float` and `input_float` read the register at the address as the high word and the next one as the low word, and decode the four bytes as an IEEE 754 single-precision number.
- **A device that sends the low word first** needs the registers the other way round. Read them as `array_int` (or `array_input_int`) with quantity 2, and decode them in a formula that swaps them: `mbFloat([FT_310_REGS[1], FT_310_REGS[0]], 0)`. `mbFloat` takes the first item of the list as the high word.
- **A 32-bit whole number**, such as a totaliser, is two registers read with quantity 2. High word first: `FQ_501_REGS[0] * 65536 + FQ_501_REGS[1]`; low word first, swap the two indexes.
- **A forwarder of type `int`** sends the whole part of the number (12.7 is sent as 12), and only its lowest 16 bits: -1 is sent as 65535, which a signed device reads as -1 again, while 70000 arrives as 4464. Keep the value between -32768 and 65535. A true or false value is sent as 1 or 0.
- **A forwarder of type `bool`** needs a variable that holds a boolean. A number is not accepted, and the write fails.

Modbus TCP Master

Reads a Modbus TCP device, or a gateway to a serial line, into variables on a schedule, and writes coils and holding registers when variables change. Reads are **collectors**; writes are **forwarders**.

Settings

Key	Setting	Type	Default	Meaning
`host`	Host	text	`localhost`	Name or IP address of the device
`port`	Port	int	`502`	Its TCP port
`unitID`	Unit ID	int	`1`	The unit identifier sent in every request. Many devices ignore it; a gateway uses it to choose the device behind it
`collectors`	Collector	list of rows	empty	The reads, one row each
`forewarders`	Forwarder	list of rows	empty	The writes, one row each. The stored key is spelled `forewarders`

Collectors

Key	Setting	Type	Default	Meaning
`field`	Variable	variable code	none	The variable the value is written into
`cron`	CRON	Quartz cron	`0 * * * * ?`	When to read. The default is every minute; the syntax is in Settings every module shares
`remoteType`	Type	register type	none	What to read, from Register types. A row without one reads nothing
`remoteAddress`	Start	int	`0`	The address of the first register or bit, counted from 0
`quantity`	Quantity	int	`0`	How many registers or bits to read, for the four array types only. The editor shows it for them alone

Forwarders

Key	Setting	Type	Default	Meaning
`field`	Variable	variable code	none	The variable whose value is written. Each change of its value sends one write
`remoteType`	Type	`bool` or `int`	none	FC05 for one coil, FC06 for one holding register
`remoteAddress`	Register	int	`0`	The address of the coil or register, counted from 0

Polling, retries and timeouts

- **Opening.** Every row's CRON is checked first: one that is empty or cannot be read stops the module from opening. Then every collector is read once, every forwarder writes its variable's current value once, and from then on each collector reads on its own CRON.
- **One request per connection.** Each request opens its own TCP connection, sends one request and closes it. A module's requests run one at a time, so a forwarder's write waits for a read already in progress.
- **Timeouts.** The answer must begin within about one second. There is no setting for it. The connection attempt has no limit of its own: an address where nothing answers holds the request for as long as the operating system takes to give up.
- **Retries.** A request that fails, an exception answer included, is tried up to three times. Between attempts a read waits a random 0.5 to 2 seconds; a write that got no answer waits 250 ms.
- **Status.** The module goes to `ERROR` only when every collector has failed `errorAfterFailures` rounds in a row. While some collectors answer, it is a `WARNING`.
- **Exceptions are not failures.** A device that answers a collector with an exception is there and working. The exception is reported as a warning about that collector and never counted. The warning stays while that collector keeps getting an exception, whatever the other collectors do, and ends when that collector reads again; with several in exception, it names the latest one. Before engine 6.18.8 the other collectors' answers cleared it at once, so it showed only for a moment.
- **Failed writes.** A write that fails after its attempts is a warning, never counted. It is cleared by the next round in which no collector is failing; in a module without collectors it stays until the module opens again.

> **Warning**
>
> A forwarder writes on the thread that changed its variable. When that variable is a formula's result, every other formula waits while the write runs: up to three attempts of about a second each against a device that does not answer, and longer when its address cannot be reached at all. Keep formula-fed forwarders on devices that are reliably there.

> **Tip**
>
> Give every forwarded variable an initial value, or make it persistent. Each forwarder writes when the module opens, and a variable that holds no value cannot be written: the attempt ends in a warning.

Messages

A counted failure is a `WARNING` that reads `… (1 of 3)`, `… (2 of 3)` and becomes an `ERROR` reading `… after 3 attempts`, with the thresholds set in Settings every module shares. The engine's own messages, such as a configuration that cannot be built, are in Module messages.

When	Level	Text
Opening: a row has no CRON	`ERROR`	`No schedule given`
Opening: a row's CRON cannot be read	`ERROR`	`Invalid schedule '{cron}': {reason}`
Reading: the connection is refused, or the attempt to connect times out	`WARNING`, then `ERROR`	`Connection refused by {address}`
Reading: the request gets no answer	`WARNING`, then `ERROR`	`No response from {address}`
Reading: the answer fails its checksum	`WARNING`, then `ERROR`	`Bad checksum from {address}`
Reading or writing: the device answers with an exception	`WARNING`	`Modbus exception {code} ({meaning}) at register {register}`
Writing: the write gets no answer	`WARNING`	`No response from {address} writing register {register}`

`{address}` is the device's **Host** and **Port**, as `host:port`, and `{register}` is the register the request was for.

The `{meaning}` names the Modbus exception: 1 illegal function, 2 illegal data address, 3 illegal data value, 4 slave device failure, 5 acknowledge, 6 slave device busy, 8 memory parity error, 10 gateway path unavailable, 11 gateway target failed to respond.

Example

A PLC at `192.168.10.21` whose manual lists a tank temperature as a float in holding registers 40010 and 40011, a level transmitter's signed reading in holding register 40005, and a temperature setpoint in holding register 40020:

```
Host      192.168.10.21
Port      502
Unit ID   1

Collector   Variable TT_101        CRON 0/10 * * * * ?   Type FC03 - Holding registers (float32)   Start 9
Collector   Variable DT_205_RAW    CRON 0/10 * * * * ?   Type FC03 - Holding register (int16)      Start 4
Forwarder   Variable SP_TT_101     Type FC06 - Holding register (int16)     Register 19
```

`TT_101` is read every ten seconds, and `DT_205`, a variable with the formula `if(DT_205_RAW > 32767, DT_205_RAW - 65536, DT_205_RAW)`, follows `DT_205_RAW` with its sign. When the module opens, it writes the current `SP_TT_101` to address 19; after that, every change of `SP_TT_101`, from a dashboard, a formula or a peer, sends FC06 to the same address.

Modbus Serial Master

Reads devices on a serial line, Modbus RTU over RS-485 or RS-232, into variables on a schedule, and writes coils and holding registers when variables change. There is no unit identifier for the module: every row names its own device, so one module serves every device on the line.

Settings

Key	Setting	Type	Default	Meaning
`port`	Port	text	none	The serial device, such as `/dev/ttyUSB0` or `COM3`. The editor offers the ports it finds on the server
`bauds`	Bauds	int	`9600`	The line speed. The editor offers 1200 to 115200
`bits`	Bits	int	`8`	Data bits: 7 or 8. Any other value opens the port with 8
`parity`	Parity	int	`0`	`0` None, `1` Even, `2` Odd. Any other value is None
`stopbits`	Stop bits	int	`2`	`0` One, `1` OnePointFive, `2` Two. Any other value is Two
`timeout`	Timeout [ms]	int	`1000`	How long to wait for an answer to begin
`collectors`	Collector	list of rows	empty	The reads, one row each
`forewarders`	Forwarder	list of rows	empty	The writes, one row each. The stored key is spelled `forewarders`

The line settings must match every device on the line. The Modbus serial standard frames each byte in 11 bits: with no parity it adds a second stop bit, which is why None and Two are the defaults. Many devices use Even parity with one stop bit instead.

Collectors

Key	Setting	Type	Default	Meaning
`field`	Variable	variable code	none	The variable the value is written into
`cron`	CRON	Quartz cron	`0 * * * * ?`	When to read. The default is every minute
`remoteUnit`	Device	int	`1`	The unit identifier of the device to ask
`remoteType`	Type	register type	none	What to read, from Register types. A row without one reads nothing
`remoteAddress`	Start	int	`0`	The address of the first register or bit, counted from 0
`quantity`	Quantity	int	`0`	How many registers or bits to read, for the four array types only

Forwarders

Key	Setting	Type	Default	Meaning
`field`	Variable	variable code	none	The variable whose value is written. Each change of its value sends one write
`remoteUnit`	Device	int	`1`	The unit identifier of the device to write to
`remoteType`	Type	`bool` or `int`	none	FC05 for one coil, FC06 for one holding register
`remoteAddress`	Register	int	`0`	The address of the coil or register, counted from 0

Polling, retries and timeouts

- **Opening.** Every row's CRON is checked first, as on the TCP master. The port is then opened once to find out whether it is there. A port that is missing, busy or not permitted is reported as an error, but the module still opens and keeps polling, and recovers by itself when the port answers, for example once a USB adapter is plugged back in. Then every collector is read once and every forwarder writes once, as on the TCP master.
- **The port is opened for each request** and closed after it. Every module that names the same port takes its turn, one request at a time, and a forwarder's write waits for a read in progress.
- **Timeouts.** The master waits up to `timeout` for the answer to begin, and takes the answer as complete once the line stays quiet for a short gap that follows the baud rate. A device that stays silent is reported as no response.
- **Retries and status** work as on the TCP master: up to three attempts per request, `ERROR` only when every collector has failed `errorAfterFailures` rounds in a row, exceptions and failed writes as warnings that are never counted. A bad checksum is counted, since on a serial line it usually means wiring, termination or line settings.

> **Warning**
>
> As on the TCP master, a forwarder writes on the thread that changed its variable, and a formula-fed forwarder makes every other formula wait for its write: about three times `timeout` against a device that does not answer.

Messages

Counted failures read `… (1 of 3)` and then `… after 3 attempts`, as described for the TCP master. `{port}` is the serial device; `{register}` and `{meaning}` read as on the TCP master.

When	Level	Text
Opening: a row has no CRON	`ERROR`	`No schedule given`
Opening: a row's CRON cannot be read	`ERROR`	`Invalid schedule '{cron}': {reason}`
Opening: the port opened	`INFO`	`Serial port {port} opened`
Opening, or reading: the port is held by another program	`ERROR` at opening; `WARNING`, then `ERROR` when reading	`Serial port {port} is busy`
Opening, or reading: the port does not exist	`ERROR` at opening; `WARNING`, then `ERROR` when reading	`Serial port {port} not found`
Opening, or reading: the service may not use the port	`ERROR` at opening; `WARNING`, then `ERROR` when reading	`Access denied to serial port {port}`
Reading: the device does not answer	`WARNING`, then `ERROR`	`No response on {port}`
Reading: the answer fails its checksum	`WARNING`, then `ERROR`	`Bad checksum on {port}`
Reading or writing: the device answers with an exception	`WARNING`	`Modbus exception {code} ({meaning}) at register {register}`
Writing: the write gets no answer	`WARNING`	`No response on {port} writing register {register}`

> **Tip**
>
> On Linux, the account the service runs as needs access to the serial device, usually through the group that owns it (`dialout` on Debian and Ubuntu). Without it the module reports `Access denied to serial port {port}`.

Example

Two energy meters, units 11 and 12, on one RS-485 line at 19200 baud, 8 data bits, Even parity and one stop bit. Each meter keeps its active power as a float in holding registers 40013 and 40014, and its energy counter in holding registers 40073 and 40074, high word first:

```
Port           /dev/ttyUSB0
Bauds          19200      Bits  8 bits      Parity  Even      Stop bits  One
Timeout [ms]   1000

Collector   Variable JT_401        CRON 0/15 * * * * ?   Device 11   Type FC03 - Holding registers (float32)
Collector   Variable JQ_401_REGS   CRON 0 * * * * ?      Device 11   Type FC03 - Holding registers (int16 array
Collector   Variable JT_402        CRON 0/15 * * * * ?   Device 12   Type FC03 - Holding registers (float32)
Collector   Variable JQ_402_REGS   CRON 0 * * * * ?      Device 12   Type FC03 - Holding registers (int16 array
```

The power readings arrive every 15 seconds. A variable `JQ_401` with the formula `JQ_401_REGS[0] * 65536 + JQ_401_REGS[1]` gives the first meter's counter, once a minute.

How a slave answers

Both slaves answer a remote master from four **register banks**. Each bank is a variable holding a list, and the master's address is the position in that list, counted from 0: a read of holding register 2 answers the third item.

Modbus table	Editor setting	Variable type	A master reads with	A master writes with
Coils	Coils variable	`[boolean]`	FC01	FC05, FC15
Discrete inputs	Discrete inputs variable	`[boolean]`	FC02	—
Holding registers	Holding registers variable	`[int]`	FC03	FC06, FC16
Input registers	Input registers variable	`[int]`	FC04	—

- **A bank's length is its size.** Give each bank variable a list of the right length, for example the initial value `[0, 0, 0, 0, 0, 0, 0, 0]` for eight registers. The editor's variable picker creates bank variables with the types above. Registers hold whole numbers from 0 to 65535.
- **Read-only banks are usually formulas.** A discrete-inputs variable with the formula `[LIT_301_HH, LIT_301_LL, P_301_FAULT]` shows three alarms to the master, and follows them as they change.
- **Writes go through the variable.** A master's write into coils or holding registers is written to the bank variable the way any other value is: checked against its type, saved when the variable is persistent, and then the formulas and modules that read the bank run once for the request. A value the variable's type refuses changes nothing, and the master still gets a normal answer. Keep writable banks free of formulas, which would recalculate over what the master wrote.
- **Persistent banks survive a restart.** A persistent variable starts from its saved value. A new persistent bank, with nothing saved yet, starts from its initial value, which is then saved, so give it one such as `[0, 0, 0, 0]`. A persistent bank with neither a saved list nor an initial value holds no list, and answers every request for it with exception 4. Before engine 6.18.8 a new persistent bank ignored its initial value and held no list until one was written to it.

- **Use single registers through formulas.** Other modules and formulas should read an item, such as `SCADA_SP[1]`, through a formula variable rather than the whole bank.

A request the slave cannot serve gets a Modbus exception, never silence, with one exception of its own: a request for another unit.

The request	The answer
For another unit identifier, broadcasts (unit 0) included	None, and nothing changes
A function code other than 01 to 06, 15 and 16	Exception 1, illegal function
A bank whose variable is not set in the module	Exception 2, illegal data address
Addresses past the end of the bank's list	Exception 2, illegal data address
2048 bits or more, or 128 registers or more, in one request	Exception 2, illegal data address
FC05 with a value other than FF00 (on) or 0000 (off)	Exception 3, illegal data value
A bank whose variable holds no list, such as a persistent bank with no saved list and no initial value	Exception 4, slave device failure

Modbus TCP Slave

Makes the instance a Modbus TCP server that a SCADA, an HMI or a PLC can read and write, from the register banks described in How a slave answers.

Settings

Key	Setting	Type	Default	Meaning
`port`	Port	int	`502`	The TCP port to listen on, on every network interface of the machine
`unit`	Unit	int	`1`	The unit identifier this slave answers to
`coils`	Coils variable	variable code	none	The coils bank, a `[boolean]` variable
`discreteInputs`	Discrete inputs variable	variable code	none	The discrete-inputs bank, a `[boolean]` variable
`holdingRegisters`	Holding registers variable	variable code	none	The holding-registers bank, an `[int]` variable
`inputRegisters`	Input registers variable	variable code	none	The input-registers bank, an `[int]` variable

Behaviour

- **The port is taken when the module opens**, before the other modules open. A port another program holds is an error, and the engine tries to open the module again after 5 seconds, then at doubling intervals up to 5 minutes.
- **Several masters at once.** Each connection is served on its own, and a master may keep its connection open or open one per request.
- **Clients are announced once.** The first connection from an address is reported, and an address from which nothing has arrived for `errorAfterSeconds` is reported as disconnected. The warning clears when a request arrives from that address again, on a new connection or on one it kept open; while other addresses are still missing, it names the one that went most recently.
- **Refused requests** are answered with their exception and reported as a warning, which the next request answered normally clears.

> **Caution**
>
> Modbus has no authentication. Anything that reaches the port can read every bank and write the coils and holding registers. Open the port only to the masters that need it, with a firewall or a separate network.

> **Note**
>
> On Linux, listening on a port below 1024, such as 502, needs a privilege a service account does not normally have. Without it the module reports `Cannot listen on port 502: …`. Grant the service that privilege, or use a higher port if the master can be set to it.

Messages

When	Level	Text
Opening: the port cannot be taken	`ERROR`	`Cannot listen on port {port}: {reason}`
Opening: the port is taken	`INFO`	`Listening on port {port}`
A master connects from a new address	`INFO`	`Client {address} connected`
Nothing has arrived from an address for `errorAfterSeconds`; cleared when a request arrives from it again	`WARNING`	`Client {address} disconnected`
A request is answered with an exception	`WARNING`	`Invalid request discarded: {reason}`

`{reason}` names the exception the request was answered with, in the words `{meaning}` uses on the TCP master: `illegal data address`, for example.

Example

A plant SCADA that reads four alarms and writes two setpoints, as unit 3:

```
Port                        502
Unit                        3
Discrete inputs variable    SCADA_ALARMS   [boolean], formula [LIT_301_HH, LIT_301_LL, P_301_FAULT, P_302_FAUL
Holding registers variable  SCADA_SP       [int], initial value [0, 0]
```

The SCADA reads FC02 at address 0, quantity 4, and gets the four alarms. When it writes FC06 to address 1 with the value 350, `SCADA_SP` becomes `[0, 350]`, and a variable `SP_LIT_301` with the formula `SCADA_SP[1] / 100` becomes 3.5. A read of holding register 2 or beyond is answered with exception 2, since `SCADA_SP` holds two items.

Modbus Serial Slave

Answers a Modbus RTU master on a serial port, from the register banks described in How a slave answers.

Settings

Key	Setting	Type	Default	Meaning
`port`	Port	text	none	The serial device, such as `/dev/ttyUSB0` or `COM3`. The editor offers the ports it finds on the server
`unit`	Unit	int	`1`	The unit identifier this slave answers to
`bauds`	Bauds	int	`9600`	The line speed. The editor offers 1200 to 115200
`bits`	Bits	int	`8`	Data bits: 7 or 8. Any other value opens the port with 8
`parity`	Parity	int	`0`	`0` None, `1` Even, `2` Odd. Any other value is None
`stopbits`	Stop bits	int	`2`	`0` One, `1` OnePointFive, `2` Two. Any other value is Two
`coils`	Coils variable	variable code	none	The coils bank, a `[boolean]` variable
`discreteInputs`	Discrete inputs variable	variable code	none	The discrete-inputs bank, a `[boolean]` variable
`holdingRegisters`	Holding registers variable	variable code	none	The holding-registers bank, an `[int]` variable
`inputRegisters`	Input registers variable	variable code	none	The input-registers bank, an `[int]` variable

There is no timeout setting: the slave's frame timing follows the baud rate.

Behaviour

- **The port is opened when the module opens, and held.** No other module can use it while the slave runs. A port that is missing, busy or not permitted is an error, and the engine tries to open the module again after 5 seconds, then at doubling intervals up to 5 minutes.
- **Frames that fail their checksum are dropped** without an answer, and counted: `errorAfterFailures` of them within `errorAfterSeconds` raise a warning. A good frame clears it once `errorAfterSeconds` have passed since the first bad frame of the current count; a bad frame that arrives after that starts a new count, so a line that keeps failing keeps the warning.
- **A port that disappears** while the slave runs, such as an unplugged USB adapter, is a warning at once and an error after `errorAfterSeconds`. The slave tries the port again after 1 second, doubling to 30 seconds, and carries on when it is back.
- **Refused requests** are answered with their exception, and not reported.

Messages

When	Level	Text
Opening: the port is held by another program	`ERROR`	`Serial port {port} is busy`
Opening: the port does not exist	`ERROR`	`Serial port {port} not found`
Opening: the service may not use the port	`ERROR`	`Access denied to serial port {port}`
The port opened, at opening or after it came back	`INFO`	`Serial port {port} opened`
The port disappeared	`WARNING`	`Serial port {port} lost`
The port has stayed away for `errorAfterSeconds`	`ERROR`	`Serial port {port} unavailable`
`errorAfterFailures` frames within `errorAfterSeconds` failed their checksum	`WARNING`	`Bad checksum on {n} frame(s)`

Example

A PLC on RS-485 that polls the instance as unit 5, at 9600 baud with the default line settings, and reads three pump run signals:

```
Port                     /dev/ttyUSB1
Unit                     5
Bauds                    9600      Bits  8 bits      Parity  None      Stop bits  Two
Discrete inputs variable   PLC_RUN   [boolean], formula [P_101_RUN, P_102_RUN, P_103_RUN]
```

The PLC reads FC02 at address 0, quantity 3. A request for unit 1 on the same line is left to the device that has it.

Next steps

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

This page describes Data Orchester engine 6.18.8.