JSON config for ‘slaves‘ (mapping ´• conversion ´• derived)

This describes the structure the Modbus master uses to poll data and publish variables per slave. Itâ s concise, Yunetas-style, and matches the current code paths.

Top level

```
"slaves": [
  {
    "id": 3,                      // Modbus unit id (1..247)
    "mapping":   [ /* poll plan */ ],
    "conversion":[ /* variables built from mapped registers */ ],
    "derived":   [ /* optional, computed variables (upper layer) */ ]
  }
]
```

â¢ ‘id‘ â required. Modbus slave/unit address.
â¢ ‘mapping‘ â required. What raw areas to read every cycle.
â¢ ‘conversion‘ â required. How to interpret the mapped registers into typed variables.
â¢ ‘derived‘ â optional. Post-processing on published variables (handled by your app layer; ‘c_prot_modbus_m‘ does not evaluate these).

‘mapping[]‘ â poll plan

Each item defines **one Modbus read** (function code is implied by ‘type‘). Sizes are bounded by the Modbus spec.

```
{
  "type": "holding_register",     // one of: "coil", "discrete_input", "input_register",
"holding_register"
  "address": 4096,                // base register/bit (0..65535). Decimal or "0x1000"
accepted.
  "size": 16,                     // quantity to read: bits for coils/inputs; 16-bit WORDS
for registers
  "id": "hr@4096"                 // optional label, used only for logging
}
```

Notes & rules

â¢ Addressing is **0-based** register numbers (not PLC â 4xxxxâ notation).
â¢ Size limits (enforced):
â¢ Coils / discrete inputs: ‘1..2000‘ bits.
â¢ Input / holding registers: ‘1..125‘ words.
â¢ Overlaps: if two mappings claim the same cell, the later one is **disabled** (logged as â Map OVERRIDEâ).
â¢ Polling order: by array order; all mappings of a slave form a **cycle**. After the last map, a publish event is emitted.

‘conversion[]‘ â typed variables from registers

Each item turns raw mapped cells into a typed value. Multi-word types are assembled from **consecutive registers** starting at ‘address‘.

```
{
  "id": "counter1",               // required, unique within the slave
  "type": "input_register",       // same domain you mapped
  "address": 4096,                // start register; must be inside a mapped span
  "format": "int64",              //
int16,uint16,bool,int32,uint32,int64,uint64,float,double,string
```

```
    "multiplier": 1,                // optional numeric scale (see below)
    "endian": "big endian",         // optional per-var: big endian (default) | little endian
  | big endian byte swap | little endian byte swap

    // STRING ONLY:
    "length_bytes": 31              // preferred for strings; fallback: use 'multiplier' as
  byte length if 'length_bytes' missing
  }
```

Supported formats & how theyâ re read

â¢ ‘bool‘
From a **coil**, **discrete_input**, or a 16-bit register (non-zero â true).
â¢ ‘int16‘ / ‘uint16‘
From one register at ‘address‘.
â¢ ‘int32‘ / ‘uint32‘ / ‘float‘
From two consecutive registers ‘[address, address+1]‘.
â¢ ‘int64‘ / ‘uint64‘ / ‘double‘
From four consecutive registers ‘[address .. address+3]‘.
â¢ ‘string‘
Takes **‘length_bytes‘** bytes from consecutive registers, high-byte then low-byte per register (wire order).
‘endian‘ is **ignored for strings**. NULs are turned into spaces; trailing spaces trimmed (legacy behavior).

The validator reserves the needed span (‘compound_value‘) for multi-word types and will **disable** the variable if any required register wasnâ t mapped.

‘endian‘ semantics (numerics only)

â¢ ‘"big endian"‘ (default): words in natural order (ABCD).
â¢ ‘"little endian"‘: reverse the whole 32/64-bit byte order (DCBA / HGFEDCBA).
â¢ ‘"big endian byte swap"‘: 16-bit word-swap (CDAB): common for IEEE-754 floats on Modbus.
â¢ ‘"little endian byte swap"‘: BADC.

‘multiplier‘ semantics

â¢ If ‘0.0 < multiplier < 1.0‘ â result is emitted as **real** (‘json_real‘).
â¢ If ‘multiplier >= 1‘ â integer formats keep **integer** type after scaling.
â¢ **STRING exception:** if ‘length_bytes‘ is absent, ‘multiplier‘ is treated as the **byte length**.

Examples

```
"mapping": [
  { "type":"holding_register", "address":4100, "size": 16 },   // reads 32 bytes
  { "type":"input_register",   "address":900,  "size": 10  }
],
"conversion": [
  { "id":"total_L", "type":"holding_register", "address":4100, "format":"int64",
"endian":"big endian", "multiplier":1 },
  { "id":"flow_m3h","type":"input_register",   "address":900,  "format":"float",
"endian":"big endian byte swap", "multiplier":1.0 },
  { "id":"tag",     "type":"holding_register", "address":4100, "format":"string",
"length_bytes":31 }
]
```

‘derived[]‘ â computed variables (optional, app-layer)

‘c_prot_modbus_m‘ **does not** evaluate ‘derived‘. They are intended for your upper layer to compute after each publish. Typical use cases: unit conversions,

combining integer/fraction parts, conditionals, clamping, etc.

Suggested shape (flexible; define in your app):

```
{
  "id": "total_L_text",
  "expr": "sprintf('%ld L', total_L)",   // your expression language
  "depends_on": ["total_L"],             // optional explicit deps
  "type": "string"                       // optional hint for downstream
}
```

Validation & common pitfalls

â¢ **Address coverage:** Every ‘conversion‘ span must lie entirely within at least one ‘mapping‘ span of the same ‘type‘. Otherwise it is disabled.
â¢ **Multi-word assembly:** Values are built from **consecutive registers**. Verify the deviceâ s word order and set ‘endian‘ accordingly (floats often need ‘big endian byte swap‘).
â¢ **Strings:** Donâ t use ‘endian‘. Use ‘length_bytes‘. Legacy fallback: ‘multiplier‘ as bytes.
â¢ **Overlaps:** Overlapping mappings or overlapping compound reads log an error; the later definition is disabled to avoid corruption.
â¢ **Limits:** Respect Modbus max counts (125 regs, 2000 bits). The code clamps/explains if exceeded.
â¢ **Address base:** Use raw register numbers (0..65535). Decimal or ‘"0xNNNN"‘ strings are accepted.
â¢ **Booleans from registers:** Non-zero â true. If you meant an individual bit inside a register, expose it via coils/discrete inputs or split at app-layer.

Minimal working slave (complete example)

```
{
  "id": 3,
  "mapping": [
    { "type":"holding_register", "address":4100, "size":16 },
    { "type":"input_register",   "address":900,  "size":10  }
  ],
  "conversion": [
    { "id":"total_L",  "type":"holding_register", "address":4100, "format":"int64",
"endian":"big endian",           "multiplier":1 },
    { "id":"flow_m3h", "type":"input_register",   "address":900,  "format":"float",
"endian":"big endian byte swap", "multiplier":1 },
    { "id":"tag",      "type":"holding_register", "address":4100, "format":"string",
"length_bytes":31 }
  ],
  "derived": [
    { "id":"total_m3", "expr":"total_L / 1000.0" }
  ]
}
```

Publishing behavior (for context)

At the end of each poll cycle per slave, the GClass emits:

```
{
  "slave_id": 3,
  "total_L": 140474,
  "flow_m3h": 12.34,
  "tag": "LINE A ..."
}
```