

Custom Protocol Definition Guide

Frame Assembler lets you define custom binary protocol frames via JSON files. Place your `.json` files in the `protocols/` folder (next to the executable) and click reload button — they will be loaded automatically.

Quick Start

Create a `.json` file in `<appdir>/protocols/`. The simplest protocol has a name and a list of fields:

```
1 {
2   "name": "My Protocol",
3   "description": "A simple fixed-length frame",
4   "version": "1.0",
5   "fields": [
6     { "name": "data", "type": "hex", "label": "Data (hex)", "length": 0, "default": "",
7       "order": 0 }
8   ]
9 }
```

Field Types

hex — User-editable hex input

The most common type. Users type hex bytes (e.g. `AA BB 01 02`) in a text field.

Attribute	Required	Description
name	Yes	Internal identifier (unique within protocol)
type	Yes	Must be "hex"
label	Yes	Display label shown in the UI
length	No	Fixed byte count. 0 or omit = variable length
default	No	Pre-filled hex value (e.g. "01", "AA BB")
tooltip	No	Help text shown on hover
order	No	Display order (lower = first)

```
1 { "name": "cmd", "type": "hex", "label": "Command", "length": 1, "default": "01", "order": 0
  }
```

constant — Read-only fixed bytes

Fixed bytes displayed as read-only. Useful for frame headers, sync bytes, packet type indicators.

Attribute	Required	Description
name	Yes	Internal identifier
type	Yes	Must be "constant"
value	Yes	Hex byte string (e.g. "AA 55" , "01")
tooltip	No	Help text shown on hover
order	No	Display order

```
1 { "name": "header", "type": "constant", "value": "AA 55", "order": 0 }
```

bits — Standalone bit-level value

Input a decimal (or `0x`-prefixed hex) value. The assembler packs it into `ceil(bitWidth/8)` bytes on its own.

Attribute	Required	Description
name	Yes	Internal identifier
type	Yes	Must be "bits"
label	Yes	Display label
bitWidth	Yes	Number of bits (1-64)
endian	No	"big" or "little" (default: "little") — byte order for multi-byte packing
default	No	Pre-filled value in decimal (e.g. "0") or hex (e.g. "0x0C")
tooltip	No	Help text
order	No	Display order

```
1 { "name": "freq", "type": "bits", "label": "Frequency", "bitWidth": 5, "default": "16",  
  "order": 2 }
```

The UI shows a text field with a range hint: `[5-bit, 0-31]` . User enters a value like `16` or `0x10` .

bit_group — Multiple bit fields packed into a shared word

Packs several bit fields into a single multi-byte word. Use this when fields share the same byte space (e.g., BLE opcode = OCF 10b + OGF 6b → 2 bytes).

Attribute	Required	Description
<code>name</code>	Yes	Internal identifier
<code>type</code>	Yes	Must be <code>"bit_group"</code>
<code>label</code>	Yes	Display label
<code>width</code>	Yes	Byte width of the container: <code>1</code> , <code>2</code> , or <code>4</code>
<code>endian</code>	No	<code>"big"</code> or <code>"little"</code> (default: <code>"little"</code>) — byte order of the packed word
<code>bitFields</code>	Yes	Array of child field definitions (see below)
<code>tooltip</code>	No	Help text
<code>order</code>	No	Display order

Child field attributes (inside `bitFields` array):

Attribute	Required	Description
<code>name</code>	Yes	Child field identifier
<code>label</code>	Yes	Display label
<code>bitWidth</code>	Yes	Number of bits (1–64)
<code>default</code>	No	Pre-filled value (decimal or <code>0x</code> hex)

Child fields are packed LSB-first: the **first** child occupies the least significant bits, and each subsequent child is shifted left by the sum of preceding widths.

```

1  {
2    "name": "opcode",
3    "label": "OpCode",
4    "type": "bit_group",
5    "width": 2,
6    "endian": "little",
7    "bitFields": [
8      { "name": "ocf", "label": "OCF", "bitWidth": 10, "default": "0" },
9      { "name": "ogf", "label": "OGF", "bitWidth": 6, "default": "8" }
10   ],
11   "order": 1
12 }
```

This packs `word = (ogf << 10) | ocf` into 2 bytes little-endian. The UI shows a grouped section with individual input fields for OCF and OGF, each with its own range hint.

`length_field` — Auto-computed length

Automatically calculates the total byte length of specified fields. The computed value is inserted at this position in the frame.

Attribute	Required	Description
<code>name</code>	Yes	Internal identifier
<code>type</code>	Yes	Must be <code>"length_field"</code>
<code>label</code>	Yes	Display label
<code>width</code>	No	Byte width: <code>1</code> , <code>2</code> , or <code>4</code> (default: <code>2</code>)
<code>endian</code>	No	<code>"big"</code> (default) or <code>"little"</code>
<code>over</code>	Yes	Array of field names whose byte length to sum
<code>order</code>	No	Display order

```
1  { "name": "len", "type": "length_field", "label": "Data Length", "width": 2, "endian": "big",
    "over": [ "payload" ], "order": 2 }
```

The `over` array can reference fields defined either **before or after** this field. The assembler processes all `hex` , `constant` , `bits` , and `bit_group` fields first, then computes `length_field` values. If the referenced fields are empty, the computed length will be `0` .

`crc` — Auto-computed CRC checksum

Calculates CRC over specified fields and appends the result.

Attribute	Required	Description
<code>name</code>	Yes	Internal identifier
<code>type</code>	Yes	Must be <code>"crc"</code>
<code>label</code>	Yes	Display label
<code>algorithm</code>	Yes	Algorithm identifier (see table below)
<code>over</code>	Yes	Array of field names to include in CRC calculation
<code>output</code>	No	Byte order: <code>"big"</code> (default) or <code>"little"</code> (Modbus uses <code>"little"</code>)
<code>order</code>	No	Display order

Supported algorithms:

Identifier	Width	Description
<code>sum8</code>	1 byte	Simple 8-bit sum of all bytes
<code>xor8</code>	1 byte	8-bit XOR of all bytes
<code>crc8_itu</code>	1 byte	CRC8-ITU (poly=0x07, init=0x00)
<code>crc8_rohc</code>	1 byte	CRC8-ROHC (poly=0x07, init=0xFF, reflected)
<code>crc8_maxim</code>	1 byte	CRC8-Maxim / Dallas 1-Wire (poly=0x31, init=0x00, reflected)
<code>crc16_ccitt</code>	2 bytes	CRC16-CCITT-FALSE (poly=0x1021, init=0xFFFF)
<code>crc16_modbus</code>	2 bytes	CRC16-Modbus (poly=0x8005, init=0xFFFF, reflected)
<code>crc16_maxim</code>	2 bytes	CRC16-Maxim (poly=0x8005, init=0x0000, reflected)
<code>crc16_usb</code>	2 bytes	CRC16-USB (poly=0x8005, init=0xFFFF, reflected, xorout=0xFFFF)
<code>crc32_ieee</code>	4 bytes	CRC32-IEEE (poly=0xEDB88320)

```
1 { "name": "crc", "type": "crc", "label": "CRC16", "algorithm": "crc16_modbus", "over": ["*"],
  "output": "little", "order": 4 }
```

The `over` Wildcard

The `over` array specifies which fields to include in CRC or length calculation. It supports three syntax forms:

Syntax	Meaning
<code>["*"]</code>	All fields defined before this field (in definition order)
<code>["*", "!crc"]</code>	All preceding fields except <code>crc</code>
<code>["header", "cmd", "data"]</code>	Explicit list of field names

The wildcard is expanded at load time. Field references are by **name**, so you can safely reorder fields without updating `over` lists.

Complete Examples

Modbus RTU Read Request

```
1 Frame: [Addr] [Func] [RegAddr 2B BE] [Count 2B BE] [CRC16 LE]
```

```

1  {
2      "name": "Modbus RTU",
3      "description": "Modbus RTU read holding registers request",
4      "version": "1.0",
5      "fields": [
6          { "name": "addr", "type": "hex", "label": "Slave Address", "length": 1,
7            "default": "01", "order": 0 },
8          { "name": "func", "type": "hex", "label": "Function Code", "length": 1,
9            "default": "03", "order": 1 },
10         { "name": "regAddr", "type": "hex", "label": "Register Address", "length": 2,
11           "default": "00 00", "order": 2 },
12         { "name": "regData", "type": "hex", "label": "Count / Data", "length": 0,
13           "default": "00 01", "order": 3 },
14         { "name": "crc", "type": "crc", "label": "CRC16 (Modbus)", "algorithm":
15           "crc16_modbus", "over": ["*"], "output": "little", "order": 4 }
16     ]
17 }

```

BLE HCI Command (H4 transport, bit_group opcode)

1 Frame: [01] [OCF 10b | OGF 6b -> 2B LE] [ParamLen 1B] [Params...]

```

1  {
2      "name": "BLE HCI Command",
3      "description": "BLE HCI Command packet (H4 transport). OpCode = OCF(10b) | OGF(6b)<<10,
4      LE",
5      "version": "1.0",
6      "fields": [
7          { "name": "pktType", "type": "constant", "value": "01",
8            "order": 0 },
9          {
10             "name": "opcode",
11             "label": "OpCode",
12             "type": "bit_group",
13             "width": 2,
14             "endian": "little",
15             "bitFields": [
16                 { "name": "ocf", "label": "OCF", "bitWidth": 10, "default": "0" },
17                 { "name": "ogf", "label": "OGF", "bitWidth": 6, "default": "8" }
18             ],
19             "tooltip": "LE Controller commands: OGF=0x08. LE Set Scan Enable: OCF=0x00C",
20             "order": 1
21         },
22         { "name": "paramLen", "type": "length_field", "label": "Parameter Length", "width":
23           1, "endian": "big", "over": ["params"], "order": 2 },
24         { "name": "params", "type": "hex", "label": "Parameters", "length": 0,
25           "default": "01", "order": 3 }
26     ]
27 }

```

Register Config (bits + bit_group demo)

```
1  Frame: [Sync AA] [RegAddr 12b BE] [Mode 3b] [Freq 5b] [Enable 1b] [CRC16]
```

```
1  {
2      "name": "Register Config",
3      "description": "Register config command demonstrating bits and bit_group types",
4      "version": "1.0",
5      "fields": [
6          { "name": "sync",      "type": "constant", "value": "AA",
7            "order": 0 },
8          { "name": "regAddr", "type": "bits",      "label": "Register Address", "bitWidth":
9            12, "endian": "big",      "default": "0", "order": 1 },
10         { "name": "mode",      "type": "bits",      "label": "Operating Mode",      "bitWidth": 3,
11           "endian": "big",      "default": "1", "order": 2 },
12         { "name": "freq",      "type": "bits",      "label": "Frequency Select", "bitWidth": 5,
13           "endian": "big",      "default": "16", "order": 3 },
14         { "name": "enable",    "type": "bits",      "label": "Enable",              "bitWidth": 1,
15           "endian": "big",      "default": "1", "order": 4 },
16         { "name": "crc",       "type": "crc",        "label": "CRC16",                  "algorithm":
17           "crc16_ccitt", "over": ["*"], "output": "big", "order": 5 }
18     ]
19 }
```

Length-Prefixed Frame with CRC

```
1  Frame: [AA BB] [Cmd 1B] [Len 2B BE] [Payload...] [CRC16]
```

```
1  {
2      "name": "Length-Prefixed Frame",
3      "description": "Header + auto-length + payload + CRC16",
4      "version": "1.0",
5      "fields": [
6          { "name": "hdr",      "type": "constant",      "value": "AA BB",
7            "order": 0 },
8          { "name": "cmd",      "type": "hex",            "label": "Command",      "length": 1,
9            "default": "01",    "order": 1 },
10         { "name": "dataLen",  "type": "length_field",    "label": "Data Length",  "width": 2,
11           "endian": "big", "over": ["payload"], "order": 2 },
12         { "name": "payload",  "type": "hex",            "label": "Payload",      "length": 0,
13           "default": "",       "order": 3 },
14         { "name": "crc",      "type": "crc",            "label": "CRC16 (CCITT)", "algorithm":
15           "crc16_ccitt", "over": ["*"], "output": "big", "order": 4 }
16     ]
17 }
```

Fixed-Length Frame with CRC32

```
1  Frame: [AA 55] [Cmd 1B] [Payload...] [55] [CRC32 BE]
```

```

1  {
2      "name": "Fixed Length Frame",
3      "description": "Sync header + command + payload + footer + CRC32",
4      "version": "1.0",
5      "fields": [
6          { "name": "sync",      "type": "constant",  "value": "AA 55",
7            "order": 0 },
8          { "name": "cmd",      "type": "hex",        "label": "Command",  "length": 1,
9            "default": "01",    "order": 1 },
10         { "name": "payload",  "type": "hex",        "label": "Payload",  "length": 0,
11           "default": "",      "order": 2 },
12         { "name": "footer",   "type": "constant",  "value": "55",
13           "order": 3 },
14         { "name": "crc",      "type": "crc",        "label": "CRC32",    "algorithm":
15           "crc32_ieee", "over": ["*"], "output": "big", "order": 4 }
16     ]
17 }

```

Rules and Limitations

1. **Field names must be unique** within a protocol.
2. **At least one field** is required.
3. **constant** fields are displayed as read-only — users can see the hex bytes but cannot edit them.
4. **length_field** can reference any field (before or after itself). The assembler processes **hex** + **constant** + **bits** + **bit_group** first, then computes lengths and CRCs.
5. **crc** must reference a registered algorithm. Supported: **crc16_ccitt**, **crc16_modbus**, **crc32_ieee**.
6. **bits** fields produce independent bytes — each field occupies its own **ceil(bitWidth/8)** bytes. Use **bit_group** when you need multiple bit fields packed into the same word.
7. **bit_group** child fields are packed **LSB-first** — the first child occupies the least significant bits.
8. Files with parse errors are silently skipped at startup — check the debug log for details.
9. File name does not matter — the protocol **"name"** field is what appears in the UI dropdown.

Troubleshooting

Symptom	Likely Cause
Protocol not in dropdown	JSON syntax error, or file not in <code>protocols/</code> folder
Field shows wrong byte count	Check <code>length</code> attribute matches actual hex input
Bits field range hint is wrong	Check <code>bitWidth</code> attribute
Bit group fields don't pack correctly	Check child field order (first = LSB) and <code>bitWidth</code> values
CRC result doesn't match expected	Verify CRC <code>algorithm</code> , <code>over</code> fields, and <code>output</code> endian
Length field shows 0	The referenced <code>over</code> fields are empty or not found
"Unknown CRC algorithm" error	Use one of: <code>sum8</code> , <code>xor8</code> , <code>crc8_itu</code> , <code>crc8_rohc</code> , <code>crc8_maxim</code> , <code>crc16_ccitt</code> , <code>crc16_modbus</code> , <code>crc16_maxim</code> , <code>crc16_usb</code> , <code>crc32_ieee</code>