Rinho Protocol Integration Guide

> **Version:** 1.2
> **Date:** 2026-05-06
> **Target audience:** Flespi engineering team / protocol integration AI
> **Protocol family:** TAIP (Trimble ASCII Interface Protocol) with Rinho extensions
> **Transport:** UDP and TCP

Table of Contents

1. Overview

Rinho devices use a protocol based on **TAIP (Trimble ASCII Interface Protocol)** with proprietary extensions. The protocol supports:

- **ASCII messages** -- Human-readable position reports, version queries, keep-alive, commands
- **Bidirectional communication** -- Server can send configuration commands and receive confirmations
- **Message numbering** -- Reliable delivery with ACK/retransmission mechanism
- **Extended telemetry** -- Temperature sensors, backup battery, hour meter, iButton, CAN bus (OBD-II/J1939)

Device Communication Flow

1. Device sends a **position report** (e.g., RCQ) with a message number (`#MSGNUM`).
2. Server receives, persists, and responds with an **ACK** echoing the same `#MSGNUM`.
3. If no ACK is received within 7 seconds, the device **retransmits** (up to 4 retries).
4. Server can send a **command** to the device with its own `#MSGNUM`.
5. Device executes the command and responds with a **confirmation** where `msgNum = original | 0x8000`.
6. Device sends periodic **keep-alive** (KA) messages; these require no ACK.

---

2. Connection & Transport

Property	Value
Transport	**UDP** and **TCP**
Default ports	Configurable (typically 4031 primary, 4034 secondary)
Encoding	ASCII (7-bit)
Max packet size	~4096 bytes (practical limit)
Keep-alive	Device sends `>KA;ID=XXXX;*CS<` every 60 seconds (configurable)

UDP Mode

- Connectionless -- device sends from dynamic port.
- Server must maintain a **NAT table** (deviceId to IP:port mapping) from the source address of the last received packet.
- Commands are sent back to the device's last-known IP:port through the **same UDP socket** that received the device's last message.

TCP Mode

- The device opens a persistent TCP connection to the server.
- **Framing is identical** to UDP: messages are delimited by `>` (start) and `<` (end).
- Since TCP is a stream protocol, the parser must **scan for `>...<` boundaries** in the byte stream. Multiple messages may arrive in one TCP segment, or one message may span multiple segments. Buffer incoming bytes and extract complete frames.
- Commands are sent back through the **same TCP connection** the device established.
- No NAT table is needed for TCP since the connection is persistent.
- The device will attempt to reconnect if the connection drops.

NAT Table (UDP only)

Since UDP is connectionless, the server must track each device's last-known IP:port to send commands back. Update this mapping every time a packet is received from a device. Consider expiring entries that have not been seen for an extended period (e.g., 24 hours).

---

3. Message Framing

All TAIP messages follow this structure:

```
>BODY[;#MSGNUM];ID=DEVICEID;*CHECKSUM<
```

Component	Required	Format	Description
`>`	Yes	Char	Start delimiter
`BODY`	Yes	ASCII	Message content (varies by type)
`;#MSGNUM`	Optional	`;#` + 4 hex digits	Message number (uppercase, e.g., `#0042`)
`;ID=DEVICEID`	Yes	`;ID=` + alphanumeric/hyphens	Device identifier (typically the IMEI)
`;*CHECKSUM`	Yes	`;*` + 2 hex digits	XOR checksum (uppercase)
`<`	Yes	Char	End delimiter

Multiple messages per packet: A single UDP datagram or TCP segment can contain multiple TAIP messages concatenated. Always scan for ALL `>...<` frames in the received data.

Example:

```
>RCQ28060426153025-3462000-05838000045180A203128000000110000050A100512;#0001;ID=860012345678901;*5E<
```

Handling Malformed Messages

- If checksum validation fails, **discard the message silently** (do not ACK).
- If the message body is too short or fields cannot be parsed, discard and log for debugging.
- Unknown message types (body does not match any known prefix) should be logged but not ACKed.

---

4. Checksum Calculation

XOR of all bytes from `>` to `*` (inclusive). The two hex digits of the checksum itself and the closing `<` are **excluded** from the calculation.

Algorithm:

1. Take the partial message from `>` through `*` (inclusive): `>ACK;#0001;ID=860012345678901;*`
2. XOR every byte in that string together: `'>' ^ 'A' ^ 'C' ^ 'K' ^ ';' ^ '#' ^ '0' ^ ... ^ ';' ^ '*'`
3. Format the result as 2 uppercase hex digits: `"5E"`
4. Append the checksum and closing delimiter: `>ACK;#0001;ID=860012345678901;*5E<`

Pseudocode:

```
def calculate_checksum(partial_message: str) -> str:
    """partial_message includes '>' through '*' (inclusive).
    The checksum digits and '<' are NOT included."""
    checksum = 0
    for byte in partial_message.encode('ascii'):
        checksum ^= byte
    return f"{checksum:02X}"
```

---

5. ASCII Position Reports (RCQ family)

Message Format

```
>R{TYPE}{66 chars base data}[extended suffix][;extra segments];#MSGNUM;ID=DEVICEID;*CHECKSUM<
```

The **base data** after the 3-char prefix (`R` + 2-char type code) is always **66 characters** with fixed positional fields.

Base Fields (66 characters, positional)

Offset	Length	Field	Format	Description	Conversion
0-1	2	ReportID	Hex `AA`	Report sequence (00-FF)	As-is
2-3	2	Day	Decimal `DD`	Day of month	Integer
4-5	2	Month	Decimal `MM`	Month (01-12)	Integer
6-7	2	Year	Decimal `YY`	Year (00-99)	2000 + YY
8-9	2	Hour	Decimal `HH`	Hour UTC (00-23)	Integer
10-11	2	Minute	Decimal `MM`	Minute (00-59)	Integer
12-13	2	Second	Decimal `SS`	Second (00-59)	Integer
14-21	8	Latitude	Signed `+NNNNNNN` or `-NNNNNNN`	Raw latitude	**value / 100000** = degrees
22-30	9	Longitude	Signed `+NNNNNNNN` or `-NNNNNNNN`	Raw longitude	**value / 100000** = degrees
31-33	3	Speed	Decimal `NNN`	Speed	**km/h** (direct, NOT knots)
34-36	3	Course	Decimal `CCC`	Heading	Degrees (0-359)
37-38	2	Inputs	Hex `HH`	Digital inputs bitmask	See Input Bitmask below
39-40	2	Outputs	Hex `II`	Digital outputs bitmask	See Output Bitmask below
41-43	3	Voltage	Decimal `JJJ`	Main power supply	**value / 10** = Volts
44-51	8	Odometer	Hex `KKKKKKKK`	Total distance	Hex to decimal = **meters**
52	1	GPSPower	`0` or `1`	GPS module power	0=Off, 1=On
53	1	GPSMode	`2` or `3`	GPS fix mode	2=2D, 3=3D
54-55	2	PDOP	Decimal `NN`	Position dilution of precision	Integer (lower = better, range 0-99)
56-57	2	Satellites	Decimal `OO`	Satellite count	Integer
58-61	4	GPSAge	Hex `PPPP`	Time since last fix	Hex to decimal = **seconds**
62	1	ModemPower	`0` or `1`	Modem power	0=Off, 1=On
63	1	GSMStatus	`0` - `5`	Registration status	See table below
64-65	2	CSQ	Decimal `SS`	Signal strength	0-30 dBm, 99=no signal

Input Bitmask (2 hex chars = 8 bits)

Bit	Name	Description
7	IGN	**Ignition** (1=On, 0=Off) -- most important bit
6	IN06	Digital input 6
5	IN05	Digital input 5
4	IN04	Digital input 4
3	IN03	Digital input 3
2	IN02	Digital input 2
1	IN01	Digital input 1
0	IN00	Digital input 0

Ignition extraction: `ignition = (inputs_byte & 0x80) != 0`

Output Bitmask (2 hex chars = 8 bits)

Bit	Name	Description
0	XP00	Digital output 0
1	XP01	Digital output 1
2	XP02	Digital output 2
3-7	--	Reserved (unused)

GSM Registration Status

Value	Meaning
0	Not registered
1	Registered (home)
2	Searching
3	Denied
4	Unknown
5	Roaming

Parsing Example

```
Raw: >RCQ28060426153025-3462000-
05838000045180A203128000000110000050A100512;#0001;ID=860012345678901;*5E<

Prefix:      R + CQ (standard position report)
Data[0:2]:   "28"          -> ReportID = 0x28
Data[2:14]:  "060426153025" -> 2026-04-06 15:30:25 UTC
Data[14:22]: "-3462000"    -> Latitude  = -3462000 / 100000 = -34.62000 degrees
Data[22:31]: "-05838000"   -> Longitude = -5838000 / 100000 = -58.38000 degrees
Data[31:34]: "045"         -> Speed     = 45 km/h
Data[34:37]: "180"         -> Course    = 180 degrees
Data[37:39]: "A2"          -> Inputs    = 0xA2 = 10100010 -> IGN=ON, IN05=ON, IN01=ON
Data[39:41]: "03"          -> Outputs   = 0x03 = XP00=ON, XP01=ON
Data[41:44]: "128"         -> Voltage   = 128 / 10 = 12.8 V
Data[44:52]: "00000011"    -> Odometer  = 0x11 = 17 meters
Data[52:53]: "0"           -> GPS Power = Off
Data[53:54]: "3"           -> GPS Mode  = 3D
Data[54:56]: "05"          -> PDOP      = 5
Data[56:58]: "10"          -> Satellites = 10
Data[58:62]: "0512"        -> GPS Age   = 0x0512 = 1298 seconds
Data[62:63]: "1"           -> Modem     = On
Data[63:64]: "1"           -> GSM       = Registered (home)
Data[64:66]: "15"          -> CSQ       = 15 dBm
MsgNum:      "0001"
DeviceID:    "860012345678901"
Checksum:    "5E"
```

---

6. Extended Report Types

All extended types share the same 66-char base structure. After the base, each type appends specific fields as a suffix.

Type Codes

Type Code	Full prefix	Description	Extra fields after base
CQ	RCQ	Standard position (filtered GPS)	None
CP	RCP	Position (unfiltered GPS, raw fix)	None
CR	RCR	Position + 1 temperature	7 chars
CV	RCV	Position + 2 temperatures	14 chars
CT	RCT	Position + iButton driver ID	17 chars (`;` + 16 hex)
CU	RCU	Position + session status + code	2+ chars (status + `;` + code)
BQ	RBQ	Position + backup battery	3 chars
BR	RBR	Position + backup + 1 temp	10 chars
BV	RBV	Position + backup + 2 temps	17 chars
HQ	RHQ	Position + backup + hour meter	11 chars
HR	RHR	Position + backup + hour meter + 1 temp	18 chars
HV	RHV	Position + backup + hour meter + 2 temps	25 chars
EQ	REQ	Position + OBD-II CAN data	Variable (separate segment)
ER	RER	Position + J1939 CAN data	Variable (separate segment)

Note on CQ vs CP: CQ applies GPS quality filtering (minimum satellites, max PDOP). CP transmits the raw GPS fix without filtering. Both share the same 66-char base structure with no extra suffix.

Extended Suffix Formats

CR -- 1 Temperature (7 chars after base)

Offset	Length	Field	Format	Conversion
0-4	5	Temp0Raw	Signed decimal `+NNNN` or `-NNNN`	**value / 10** = degrees Celsius
5-6	2	Temp0Age	**Hex** `HH`	Hex to decimal = seconds since last reading

Example: `+0235` = +23.5 C, age `0A` = 10 seconds ago

Important: The temperature age field is hexadecimal (0x00-0xFF = 0-255 seconds), not decimal.

CV -- 2 Temperatures (14 chars)

Offset	Length	Field
0-4	5	Temp0Raw (signed decimal)
5-6	2	Temp0Age (hex)
7-11	5	Temp1Raw (signed decimal)
12-13	2	Temp1Age (hex)

CT -- iButton (17 chars)

Offset	Length	Field	Description
0	1	Separator	Literal `;` character (always present)
1-16	16	iButton ID	Dallas 1-Wire DS1990 ROM ID, 8 bytes as 16 hex characters

This is a standard Dallas/Maxim 1-Wire 64-bit identifier used for driver identification. The 8 bytes include: 1 family code byte + 6 serial bytes + 1 CRC byte.

Important: The CT suffix begins with a literal `;` character before the 16 hex chars. Skip the leading `;` when extracting the ID.

CU -- Session (variable, 2+ chars)

Offset	Length	Field	Values
0	1	Status	`0` = closed (driver logged out), `1` = open (driver logged in)
1	1	Separator	Literal `;` character
2+	var	SessionCode	Operator/driver code identifying the session (alphanumeric, length depends on configuration)

Example: `1;DRIVER01` = session open with operator code `DRIVER01`. A "session" represents a tracked work period. The device tracks accumulated time, distance, and engine hours per session.

BQ -- Backup Battery (3 chars)

Offset	Length	Field	Conversion
0-2	3	BackupVoltageRaw	Decimal, **value / 100** = Volts

Example: `416` = 4.16 V. This is the device's internal backup battery voltage.

BR -- Backup + 1 Temperature (10 chars)

Offset	Length	Field
0-2	3	BackupVoltageRaw
3-7	5	Temp0Raw
8-9	2	Temp0Age (hex)

BV -- Backup + 2 Temperatures (17 chars)

Offset	Length	Field
0-2	3	BackupVoltageRaw
3-7	5	Temp0Raw
8-9	2	Temp0Age (hex)
10-14	5	Temp1Raw
15-16	2	Temp1Age (hex)

HQ -- Backup + Hour Meter (11 chars)

Offset	Length	Field	Conversion
0-2	3	BackupVoltageRaw	Decimal, **value / 100** = Volts
3-10	8	HorometerRaw	Hex to decimal = **seconds** of engine runtime

Example: `4160003 6EE0` → backup `416` = 4.16 V, horometer `00036EE0` = 0x36EE0 = 225,000 seconds = 62.5 hours.

HQ includes the device's internal backup battery voltage followed by the engine hour meter. For hour meter with backup voltage **plus** temperature sensors, use HR (1 temp) or HV (2 temps).

HR -- Backup + Hour Meter + 1 Temperature (18 chars)

Offset	Length	Field
0-2	3	BackupVoltageRaw
3-10	8	HorometerRaw
11-15	5	Temp0Raw
16-17	2	Temp0Age (hex)

HV -- Backup + Hour Meter + 2 Temperatures (25 chars)

Offset	Length	Field
0-2	3	BackupVoltageRaw
3-10	8	HorometerRaw
11-15	5	Temp0Raw
16-17	2	Temp0Age (hex)
18-22	5	Temp1Raw
23-24	2	Temp1Age (hex)

EQ/ER -- CAN Bus Data (Variable)

CAN data is **not** part of the positional suffix -- it appears as a separate `;`-delimited segment in the packet. See Section 15.

---

7. Version Report (RVR)

Device firmware version query response.

```
>RVR {VERSION_STRING};[#MSGNUM];ID=DEVICEID;*CHECKSUM<
```

Field	Description
VERSION_STRING	Free-form string identifying firmware and hardware variant
MSGNUM	Present when responding to a version query command (will have bit 15 set)

Real-world examples:

```
>RVR RINHO IOT v1.09.20 SM BG95 LC86G 8MB;#8042;ID=860012345678901;*CD<
>RVR RINHO IOT v1.09.20 ZE EG915U LC86G 4MB;#8042;ID=860012345678901;*AB<
>RVR RINHO IOT v1.09.20 SP BG96 GPS96 8MB;#8042;ID=860012345678901;*EF<
```

Version string format: `{PRODUCT} v{VERSION} {MODEL} {MODEM} {GPS} {FLASH}`

Component	Values	Description
PRODUCT	`RINHO IOT`	Product name
VERSION	`1.09.20`	Firmware version (major.minor.patch)
MODEL	`SM`, `ZE`, `SP`	Device model: SM=Smart, ZE=Zero, SP=Spider
MODEM	`BG95`, `BG96`, `UG96`, `EG915U`	Cellular modem chip
GPS	`LC86G`, `LC86L`, `GPS96`	GPS module
FLASH	`4MB`, `8MB`, `16MB`	Flash memory size

8. Keep-Alive (KA)

Heartbeat message. **No ACK is required** for keep-alive.

```
>KA;ID=DEVICEID;*CHECKSUM<
```

- Default interval: **60 seconds** (configurable per device).
- The server should update the NAT table (UDP) or track connection liveness (TCP) when receiving a KA.
- Do **not** send any response to a KA message.

---

9. Device Inventory Messages

Devices can report hardware information. These are typically responses to inventory query commands:

Prefix	Field	Example	Format
`RCXHWI`	Hardware ID	`>RCXHWI2210;#8020;ID=860012345678901;*AB<`	4 digits: modem + GPS + ADC + LEDs model codes
`RSN`	Serial Number	`>RSNAABBCCDDEEFF000000000042;#8021;ID=860012345678901;*CD<`	24 hex chars: MAC address (reversed) + padding + XOR checksum
`RIMEI`	IMEI	`>RIMEI860012345678901;#8022;ID=860012345678901;*EF<`	15-digit standard IMEI
`RTAG`	Config Tag	`>RTAGFLEET_01;#8023;ID=860012345678901;*GH<`	User-configurable label string

These will have `msgNum >= 0x8000` since they are responses to server commands (QCX HWI, QSN, QIMEI, QTAG).

---

10. ACK Mechanism

When to ACK

- ACK is required when the device sends a message **with a** `#MSGNUM` where `msgNum < 0x8000`.
- **Do NOT ACK:** keep-alive (KA) messages, messages with `msgNum >= 0x8000` (these are command responses), or messages without a `#MSGNUM`.

ACK Format

```
>ACK;#MSGNUM;ID=DEVICEID;*CHECKSUM<
```

The server echoes back the exact `MSGNUM` and `DEVICEID` from the received message.

ACK Timing (Critical)

ACK must be sent ONLY AFTER the message has been successfully persisted/processed. If persistence fails, do not ACK -- the device will retransmit. This guarantees no data loss.

Example

Device sends:

```
>RCQ28060426...;#002A;ID=860012345678901;*5E<
```

Server responds:

```
>ACK;#002A;ID=860012345678901;*XX<
```

Where `XX` is the XOR checksum of `>ACK;#002A;ID=860012345678901;*`.

---

11. Device Retransmission Behavior

If the device does not receive an ACK after sending a report:

Parameter	Value
Timeout per attempt	**7 seconds**
Maximum retries	**4**
Total window before giving up	~35 seconds
Send window size	Up to **9** messages in parallel
Behavior on exhaustion	Message is discarded after all retries fail

The device maintains a send window of up to 9 pending messages. Each message is retransmitted independently on its own timeout cycle. After 4 failed retries (no ACK received), the message is dropped.

Poison protection: If the device detects that 3+ consecutive connection resets occur without any successful ACK, it discards all pending messages to avoid sending stale data after reconnection.

Message Number (msgNum) Management

- Range: `0x0000` to `0x7FFF` (0 to 32767).
- Wrapping: after `0x7FFF`, the counter resets to `0x0000`.
- Each message gets a unique, incrementing msgNum.
- The msgNum is used to match ACKs to sent messages -- the server must echo the exact msgNum.

---

12. Command Sending (Server to Device)

Command Message Format

Commands sent from server to device use the same TAIP framing:

```
>COMMAND_BODY;#MSGNUM;ID=DEVICEID;*CHECKSUM<
```

Component	Description
COMMAND_BODY	The command string (device-specific, opaque to the server)
MSGNUM	Server-assigned, 4 hex digits (range 0x0001-0x7FFF)
DEVICEID	Target device identifier
CHECKSUM	XOR checksum of everything from `>` to `*`

Command Construction

1. Choose a `msgNum` (4 hex digits, range 0x0001-0x7FFF). Increment per command.
2. Build the partial message: `>COMMAND;#MSGNUM;ID=DEVICEID;*`
3. Calculate XOR checksum of that string.
4. Append checksum + `<`.

Example -- requesting firmware version:

```
cmd = ">QVR;#0042;ID=860012345678901;*"
checksum = calculate_checksum(cmd)  # e.g., "3F"
full_cmd = cmd + checksum + "<"     # ">QVR;#0042;ID=860012345678901;*3F<"
```

Command Body

The command body is **opaque** -- it can be virtually any string that the device firmware recognizes. The device has 100+ internal commands for configuration, queries, and control. The specific command catalog is device-firmware-dependent and outside the scope of this protocol document.

What matters for the ingestion layer is:

- **Framing** is always the same: `>BODY;#MSGNUM;ID=DEVICEID;*CS<`
- **Checksum** is always calculated the same way.
- **Confirmation** always follows the `msgNum | 0x8000` convention.

Unknown Commands

If the device receives a command it does not recognize, it **silently ignores it** -- no error response is sent. The server should implement a timeout to detect unacknowledged commands.

---

13. Command Confirmation Flow

Message Number Convention

Range	Direction	Meaning
`0x0000-0x7FFF`	Device to Server	Device reports (position, version, etc.)
`0x0000-0x7FFF`	Server to Device	Server commands
`0x8000-0xFFFF`	Device to Server	**Confirmation of a command** (echoes server msgNum with bit 15 set)

Confirmation Logic

When the server sends a command with `#0042`:

1. The device receives and executes it.
2. The device responds with `msgNum = 0x0042 | 0x8000 = 0x8042`.
3. The response body contains the command result (e.g., version string, config dump, or just an echo).

Example flow:

Server sends: `>QVR;#0042;ID=860012345678901;*3F<` (query firmware version)

Device responds: `>RVR RINHO IOT v1.09.20 SM BG95 LC86G 8MB;#8042;ID=860012345678901;*AB<` (response, msgNum 0x8042)

Detecting Responses

```
msg_num_int = int(msg_num_hex, 16)
is_response = msg_num_int >= 0x8000
original_cmd_num = msg_num_int & 0x7FFF  # strip bit 15 to get original
```

---

14. Additional Parameters (AP=/PA=)

Some reports include an extra `;AP=` or `;PA=` segment with custom sensor data. Both prefixes are equivalent (firmware version determines which is used). The segment contains comma-separated typed parameters generated by user-configured report scripts.

Format

```
>R{TYPE}{base...};AP={param1},{param2},...;#MSGNUM;ID=DEVICEID;*CHECKSUM<
>R{TYPE}{base...};PA={param1},{param2},...;#MSGNUM;ID=DEVICEID;*CHECKSUM<
```

Parameter Structure

Each parameter follows the format `name:type:value`, where:

Type	Meaning	Value format	Example
`1`	Integer	Whole number	`fuel:1:0`, `rpm:1:1496`
`2`	Float	Decimal number	`temp:2:28.9`, `hum:2:75.2`
`3`	String	Text (no `;` or control chars)	`status:3:ok`, `driver:3:Juan`

Parameter names are alphanumeric with underscores (`[a-zA-Z0-9_]`).

Some legacy scripts may omit the type field, producing `name:value` pairs (e.g., `c0:1496`). The ingestion layer should handle both formats.

Real-world examples

CAN data from a vehicle script (legacy format without types):

```
>RHQ03...;PA=id:0000059278,c0:1496,c2:61,c4:,c6:84,c7:;#B6BD;ID=863238070628120;*16<
```

BLE/analog sensor data (typed format):

```
>RBQ05...;AP=fuel:1:0,temp:2:28.9;#1016;ID=860012345678901;*2B<
>RCQ00...;AP=hum:2:75.2,alt:2:-15.5;ID=860012345678901;*FF<
```

Extraction

1. Search for `;AP=` in the raw packet. If not found, search for `;PA=`.
2. Extract from after the `=` until the next `;`.
3. The result is a comma-separated string of parameters that can be further split and typed if needed.

---

15. CAN Bus Data (EQ/ER)

OBD-II (EQ)

```
>REQ{66 base chars};{PID=VALUE,PID=VALUE,...};#MSGNUM;ID=DEVICEID;*CS<
```

J1939 (ER)

```
>RER{66 base chars};{SPN=VALUE,SPN=VALUE,...};#MSGNUM;ID=DEVICEID;*CS<
```

CAN data appears as a separate segment (between `;` delimiters) containing `CODE=VALUE` pairs, usually comma-separated. A segment may contain a single parameter (no comma) or multiple.

Examples:

```
>REQ...;2010=1000.00,5000=0.00;#0001;ID=860012345678901;*AB<
>REQ...;1=WVWZZZ3CZWE123456;#0002;ID=860012345678901;*CD<
```

Identifying the CAN segment: Split the packet by `;`. Skip segments that match known prefixes (`ID=`, `AP=`, `PA=`, `TXT=`, `#`, `*`) and segments starting with `R` (the report body). The CAN segment is the one containing `=`. Check first for segments with both `=` and `,` (multi-param), then fall back to segments with just `=` (single-param).

Note: The PID/SPN codes and their units are vehicle-specific and depend on the OBD-II/J1939 standard. This document does not include a PID/SPN lookup table -- refer to standard OBD-II PID references (e.g., SAE J1979) or J1939 SPN databases for interpretation.

---

16. Parsed Output Structure

After parsing a raw TAIP packet, the following structured data should be produced. This schema is implementation-agnostic -- use it to feed your internal pipeline, database, message queue, or API.

Common Fields (all message types)

Field	Type	Example	Description
`device_id`	string	`"869456012345678"`	Device identifier (typically the IMEI)
`msg_type`	string	`"position"`	Message classification (see below)
`msg_num`	string	`"002A"`	4 hex digits uppercase (empty if absent)
`is_response`	boolean	`false`	`true` if msgNum >= 0x8000 (command response)
`event_time`	datetime	`2026-04-06T15:30:25Z`	GPS timestamp from the device (UTC)
`received_at`	datetime	`2026-04-06T15:30:26Z`	Server receive time (UTC)
`raw`	string	`">RCQ...;*5E<"`	Complete original packet (for debugging/replay)

Note on timestamps: `event_time` comes from the device GPS clock and may have drift. `received_at` is the server's clock. Use `received_at` for ordering when device clock reliability is uncertain.

msg_type Values

Value	Description
`position`	GPS position report (RCQ family)
`version`	Firmware version report (RVR)
`confirmation`	Response to a server command where the body does not match a known type
`keepalive`	Heartbeat (KA)
`unknown`	Unrecognized message type (not a response)

Classification priority: First classify by body content (position, version, keepalive). Then check `is_response` (msgNum >= 0x8000). If the body matched a known type, **keep that type** and set `is_response = true`. Only use `confirmation` when `is_response = true` AND the body did not match any known type. For example, a version response `>RVR ...;#8042;...` has `msg_type = "version"` with `is_response = true`, not `msg_type = "confirmation"`.

Position Fields (when msg_type = "position")

Field	Type	Example	Description
`latitude`	float	`-34.62000`	Decimal degrees
`longitude`	float	`-58.38000`	Decimal degrees
`speed`	integer	`45`	km/h
`course`	integer	`180`	Degrees (0-359)
`ignition`	boolean	`true`	Derived from inputs bit 7

Telemetry Fields (from the 66-char base)

Field	Raw Example	Conversion	Converted
`report_type`	`"HR"`	--	Report subtype
`report_id`	`"28"`	--	Hex sequence 00-FF
`inputs`	`"A2"`	Hex bitmask	Bit 7=IGN, bits 6-0=digital inputs
`outputs`	`"03"`	Hex bitmask	Bits 0-2=digital outputs
`voltage_raw`	`"128"`	/ 10	12.8 V
`odometer_raw`	`"00000011"`	Hex to decimal	17 meters
`gps_power`	`"1"`	--	0=Off, 1=On
`gps_mode`	`"3"`	--	2=2D, 3=3D
`pdop`	`"05"`	--	Decimal integer, lower=better
`satellites`	`"10"`	--	Decimal integer
`gps_age`	`"0512"`	Hex to decimal	1298 seconds since fix
`modem_power`	`"1"`	--	0=Off, 1=On
`gsm_status`	`"1"`	--	0-5 registration status
`csq`	`"15"`	--	0-30 dBm, 99=no signal

Extended Telemetry Fields (present depending on report type)

Field	Report Types	Raw Example	Conversion
`backup_voltage_raw`	BQ, BR, BV, HQ, HR, HV	`"416"`	/ 100 = 4.16 V (internal battery)
`horometer_raw`	HQ, HR, HV	`"00036EE0"`	Hex to decimal = seconds of engine runtime
`temp0_raw`	CR, CV, BR, BV, HR, HV	`"+0235"`	/ 10 = +23.5 degrees C
`temp0_age`	CR, CV, BR, BV, HR, HV	`"0A"`	**Hex** to decimal = 10 seconds
`temp1_raw`	CV, BV, HV	`"-0253"`	/ 10 = -25.3 degrees C
`temp1_age`	CV, BV, HV	`"0B"`	**Hex** to decimal = 11 seconds
`ibutton`	CT	`"0102030405060708"`	Dallas 1-Wire DS1990 ROM ID (16 hex chars, `;` prefix in raw is stripped)
`session`	CU	`"1"`	`"0"` = closed, `"1"` = open
`session_code`	CU	`"DRIVER01"`	Operator/driver code identifying the active session
`can_protocol`	EQ, ER	`"OBD-II"`	OBD-II or J1939
`can_data`	EQ, ER	`"2010=1000.00"`	Comma-separated CODE=VALUE pairs
`ap_params`	Any	`"fuel:1:0,temp:2:28.9"`	Additional parameters from `;AP=` or `;PA=` segment

Version Fields (when msg_type = "version")

Field	Type	Example	Description
`version`	string	`"RINHO IOT v1.09.20 SM BG95 LC86G 8MB"`	Full firmware + hardware descriptor

Inventory Fields (when present in response)

Field	Type	Example	Description
`hwi`	string	`"2210"`	4 digits: modem + GPS + ADC + LEDs model codes
`sn`	string	`"AABBCCDDEEFF000000000042"`	24 hex chars derived from device MAC address
`imei`	string	`"860012345678901"`	15-digit IMEI (same as device_id)
`tag`	string	`"FLEET_01"`	User-configurable label

Full Parsed Example

Input:

```
>RHR28060426153025-3462000-
05838000045180A203128000000110000050A10051241600036EE0+02350A;#002C;ID=860012345678901;*BC<
```

Output:

```
{
  "device_id": "860012345678901",
  "msg_type": "position",
  "msg_num": "002C",
  "is_response": false,
  "event_time": "2026-04-06T15:30:25Z",

  "latitude": -34.62000,
  "longitude": -58.38000,
  "speed": 45,
  "course": 180,
  "ignition": true,

  "report_type": "HR",
  "inputs": "A2",
  "outputs": "03",
  "voltage_raw": "128",
  "odometer_raw": "00000011",
  "gps_power": "0",
  "gps_mode": "3",
  "pdop": "05",
  "satellites": "10",
  "gps_age": "0512",
  "modem_power": "1",
  "gsm_status": "1",
  "csq": "15",

  "backup_voltage_raw": "416",
  "horometer_raw": "00036EE0",
  "temp0_raw": "+0235",
  "temp0_age": "0A"
}
```

17. Device Models

There are three hardware variants: **Rinho Zero** (ZE), **Rinho Spider** (SP), and **Rinho Smart** (SM). They differ in hardware peripherals (modem, GPS, ADC), but at the protocol level they are identical.

Any model can generate any report type. The report type is configured per-device (via scripts or commands), not determined by the hardware model. A Rinho Zero can send RHV reports just as a Rinho Smart can send plain RCQ reports. The parser should **not** filter or restrict report types based on the device model.

All models support:

- All ASCII position report types (CQ, CP, CR, CV, CT, CU, BQ, BR, BV, HQ, HR, HV, EQ, ER)
- Version query (QVR produces RVR)
- Keep-alive (KA)
- Configurable report intervals and event triggers
- Ignition detection (input bit 7)
- Bidirectional commands with confirmation

---

18. Security Considerations

The TAIP protocol as used by Rinho devices has the following security characteristics:

- **No encryption.** All data is transmitted in plaintext over UDP/TCP.
- **ID-based identification only.** There is no cryptographic authentication between device and server. Any source that knows a device's ID could potentially send spoofed packets.
- **Device-level password.** Devices support a password for configuration commands (SPW/QPW), but this is for device management, not for report authentication.

Recommendation for integrators: Use network-level security (VPN, IP whitelisting, firewall rules) to restrict which sources can send data to the ingestion port. Consider validating device IDs against a known device registry.

---

19. Integration Checklist

- [ ] **UDP and/or TCP listener** on configurable port
- [ ] **Frame scanning:** find all `>...<` delimited messages in received data
- [ ] **TCP stream buffering:** buffer incoming bytes and extract complete `>...<` frames across segment boundaries
- [ ] **Checksum validation** (XOR of bytes from `>` through `*`, compare with 2 hex digits after `*`)
- [ ] **ASCII parser:** extract all fields from the 66-char positional base
- [ ] **All extended report types:** CR, CV, CT, CU, BQ, BR, BV, HQ, HR, HV, EQ, ER
- [ ] **ACK sending** after successful processing (not for KA, not for msgNum >= 0x8000)
- [ ] **NAT table** maintenance for UDP (deviceId to IP:port)
- [ ] **Message numbering:** detect responses (msgNum >= 0x8000)
- [ ] **RVR** (firmware version) parsing
- [ ] **Device inventory messages:** RCXHWI, RSN, RIMEI, RTAG
- [ ] **Command sending** with checksum construction
- [ ] **Command confirmation matching** (msgNum | 0x8000)
- [ ] **CAN bus data extraction** (EQ/ER segments)
- [ ] **Additional parameters extraction** (`;AP=` / `;PA=` segments)
- [ ] **Keep-alive handling** (NAT refresh, no ACK)

Common Pitfalls

1. **Speed is km/h** -- NOT knots. Do not multiply by 1.852.
2. **Latitude divisor is 100000** -- yields decimal degrees (e.g., -3462000 = -34.62000 degrees).
3. **Longitude divisor is 100000** -- same (e.g., -05838000 = -58.38000 degrees).
4. **PDOP is decimal** -- the 2-char field is a decimal integer (0-99), not hex.
5. **Temperature age is hex** -- the 2-char age field IS hex (0x00-0xFF = 0-255 seconds).
6. **Odometer is in meters** -- hex string, convert to decimal.
7. **Voltage is in tenths** -- `128` = 12.8V.
8. **ACK before persistence = data loss** -- always persist first, then ACK.
9. **MsgNum 0x8000+ is a response** -- do not ACK, do not treat as a new report.
10. **Multiple packets per datagram** -- always scan for ALL `>...<` frames in received data.

Appendix A: Sample Messages

Standard Position (CQ)

```
>RCQ28060426153025-3462000-05838000045180A203128000000110000050A100512;#002A;ID=860012345678901;*5E<
```

Position with Temperature (CR)

```
>RCR28060426153025-3462000-
05838000045180A203128000000110000050A100512+02350A;#002B;ID=860012345678901;*AF<
```

Position with Backup + Hour Meter (HQ)

```
>RHQ28060426153025-3462000-
05838000045180A203128000000110000050A10051241600036EE0;#002B;ID=860012345678901;*BC<
```

Sufijo `4160003 6EE0` (sin espacio): backup `416` = 4.16 V + horometer `00036EE0` = 62.5 hours.

Position with Backup + Hour Meter + 2 Temps (HV)

```
>RHV28060426153025-3462000-05838000045180A203128000000110000050A10051241600036EE0+02350A-
02530B;#002C;ID=860012345678901;*BC<
```

Position with iButton (CT)

```
>RCT28060426153025-3462000-
05838000045180A203128000000110000050A100512;01ABCDEF12345678;#002D;ID=860012345678901;*AA<
```

Note the literal `;` separator before the 16-hex iButton ID.

Position with Session (CU)

```
>RCU28060426153025-3462000-
05838000045180A203128000000110000050A1005121;DRIVER01;#002E;ID=860012345678901;*AB<
```

Status `1` (open) followed by `;` and operator code `DRIVER01`.

Version Response

```
>RVR RINHO IOT v1.09.20 SM BG95 LC86G 8MB;#8042;ID=860012345678901;*CD<
```

Keep-Alive

```
>KA;ID=860012345678901;*2F<
```

ACK

```
>ACK;#002A;ID=860012345678901;*45<
```

Command (Server to Device)

```
>QVR;#0042;ID=860012345678901;*3F<
```

Command Confirmation (Device to Server)

```
>RVR RINHO IOT v1.09.20 SM BG95 LC86G 8MB;#8042;ID=860012345678901;*AB<
```

CAN Data (OBD-II)

```
>REQ28060426153025-3462000-
05838000045180A203128000000110000050A100512;2010=1000.00,5000=0.00;#002D;ID=860012345678901;*DE<
```

Appendix B: Data Type Conversion Quick Reference

Raw Field	Stored As	Conversion	Unit
Latitude	`+NNNNNNN` or `-NNNNNNN`	/ 100000	Decimal degrees
Longitude	`+NNNNNNNN` or `-NNNNNNNN`	/ 100000	Decimal degrees
Speed	Decimal `NNN`	Direct	km/h
Course	Decimal `CCC`	Direct	Degrees (0-359)
Voltage	Decimal `JJJ`	/ 10	Volts
Backup Voltage	Decimal `NNN`	/ 100	Volts
Odometer	Hex `KKKKKKKK`	Hex to decimal	Meters
Hour Meter	Hex `NNNNNNNN`	Hex to decimal	Seconds (engine runtime)
Temperature	Signed decimal `+NNNN`	/ 10	Degrees Celsius
Temperature Age	**Hex** `HH`	Hex to decimal	Seconds
GPS Age	Hex `PPPP`	Hex to decimal	Seconds
PDOP	Decimal `NN`	Direct	Integer (lower = better)

This document describes the Rinho TAIP protocol v1.1. The protocol supports both UDP and TCP transport with identical message framing.