KTS TCP Socket API — Remote (External Network) Version

Overview

This API defines the **Device Host ↔ Server** TCP/IP control protocol for remote (WAN) communication. TCP Socket is used for connection stability and to avoid data-length constraints.

---

1. TCP Server Address

Domain and port are configured in the management backend.

Example: `1709-jg.shbaiche.com` — Port: `41709`

---

2. Protocol Base Format

Each message frame consists of three parts:

Part	Description
Frame Header	Device SN or MAC address (`sn`)
Command	`cmd` field
Payload	`data` field (JSON text)

Encoding: All strings use **UTF-8**

Max packet size: ≤ 40KB per datagram (based on the largest object set in a zone)

Frame delimiter: `\r\n` (0x0D 0x0A) — used to separate packets

Base Frame Structure

```
{"sn":"BC0008ABEF112233", "cmd":"xxx", "data":"xxxxxxxxxxxxxxxxx"}\r\n
```

> ⚠️ **Field ordering is strict:** `sn` → `cmd` → `data` — must follow this exact order in all frames.

---

3. Command Table

Message Name	Direction	`cmd`
Device host registration / online	Host → Server	`reg`
Registration reply	Server → Host	`regrsp`
Zone query	Server → Host	`query1`
Zone query reply	Host → Server	`rsp1`
Object query	Server → Host	`query2`
Object query reply	Host → Server	`rsp2`
Object control	Server → Host	`control`
Object control acknowledgment	Host → Server	`controlrsp`
State push (status update)	Host → Server	`push`
Modify host key / domain	Server → Host	`cfg`
Modify reply	Host → Server	`cfgrsp`
Scheduled task	Server → Host	`timing`
Scheduled task reply	Host → Server	`timingrsp`
Heartbeat (10–60s interval)	Host → Server	`keepalive`
Heartbeat reply	Server → Host	`keepalive`

---

4. Authentication & Signing

On every new TCP connection, the device host must first send a `reg` (registration) frame. This frame requires:

Field	Description
`timestamp`	Unix timestamp (seconds)
`rand`	Random number
`sign`	MD5 signature

Signature formula:

```
sign = MD5(sn + timestamp + rand + key)
```

(Assume device key = `1234567890`)

---

5. Protocol Examples

5.1 Device Registration

Host → Server (reg):

```
{"sn":"08ABEF112233","cmd":"reg","timestamp":"1508845817","rand":"35","sign":"055
```

Server → Host (regrsp):

```
{"sn":"08ABEF112233","cmd":"regrsp","data":{"error":"0"}}\r\n
```

`error`	Meaning
`"0"`	Registration successful
`"1"`	Device unauthorized / invalid

5.2 Zone Query

Server → Host (query1):

```
{"sn":"08ABEF112233","cmd":"query1","data":{"oRoom":"-1","tag":"222"}}\r\n
```

> `oRoom: "-1"` → query all zones

Host → Server (rsp1):

```
{"sn":"08ABEF112233","cmd":"rsp1","data":[
  {"nID":"1","nName":"r1","pID":"-1","Icon":"@bedroom"},
  {"nID":"2","nName":"r2","pID":"-1","Icon":"@office"}
]}\r\n
```

Field	Description
`nID`	Zone ID
`nName`	Zone name
`pID`	Parent zone ID (`-1` = root)
`Icon`	Icon reference

5.3 Object Query

Server → Host (query2):

```
{"sn":"08ABEF112233","cmd":"query2","data":{"oRoom":"1","tag":"2"}}\r\n
```

Host → Server (rsp2):

```
{"sn":"08ABEF112233","cmd":"rsp2","data":[
  {"oID":"37","EIS":"1","val":"-1","oName":"Chandelier Switch","room":"2","type":
  {"oID":"38","EIS":"2","val":"-1","oName":"Spotlight Dimmer","room":"2","type":"
  {"oID":"44","EIS":"14","val":"-1","oName":"Welcome Scene","room":"2","type":"6"
  {"oID":"45","EIS":"14","val":"-1","oName":"Leave Scene","room":"2","type":"6","
  {"oID":"46","EIS":"14","val":"-1","oName":"TV Scene","room":"2","type":"6","add
]}\r\n
```

Field	Description
`oID`	Object ID (unique identifier)
`EIS`	EIS data type (see Appendix)
`val`	Current value (`-1` = not yet read)

`oName`	Object display name
`room`	Zone ID this object belongs to
`type`	Device type (1=light, 2=curtain, 3=AC, 4=floor heat, 5=fresh air, 6=scene, 9=security)
`addrSet`	KNX group address — primary (write)
`addrSet2`	KNX group address — secondary (status feedback)
`addrSet3`	KNX group address — tertiary
`group`	Object group tag
`Icon`	Icon reference

5.4 Object Control

Server → Host (control):

```
{"sn":"08ABEF112233","cmd":"control","data":"EIS1=1/1/1=1"}\r\n
```

Control data format: `EIS_TYPE=GROUP_ADDRESS=VALUE`

Host → Server (controlrsp):

```
{"sn":"08ABEF112233","cmd":"controlrsp","data":{"error":"0"}}\r\n
```

5.5 State Push

> The `oID` must be obtained in advance via `query2`. The `push` message uses `oID` as the unique object identifier.

Host → Server (push):

```
{"sn":"08ABEF112233","cmd":"push","data":[
  {"oID":"37","EIS":"1","val":"1","room":"","navi":""}
]}\r\n
```

5.6 Modify Host Key / Domain

Server → Host (cfg):

```
{"sn":"08ABEF112233","cmd":"cfg","data":{"key":"1234567890","domain":"kanonbus.co
```

Host → Server (cfgrsp):

```
{"sn":"08ABEF112233","cmd":"cfgrsp","data":{"error":"0","key":"1234567890"}}\r\n
```

`error`	Meaning
`"0"`	Modification successful
`"1"`	Modification failed

5.7 Heartbeat

Host → Server (keepalive):

```
{"sn":"08ABEF112233","cmd":"keepalive"}\r\n
```

(Server replies with the same format)

5.8 Scheduled Task

Server → Host (timing):

```
{"sn":"08ABEF112233","cmd":"timing","data":{
  "tid":"123",
  "tname":"My Scene",
  "oRoom":"0",
  "flag":"1",
  "time":"09:55",
  "action":"1=2/3/4=0,1=2/3/5=1",
  "repeat":",1,3,4,5,6,7,"
}}\r\n
```

Field	Description

`tid`	Scheduled task ID
`flag`	`1` =enable, `0` =disable, `2` =delete
`time`	Trigger time (HH:MM, zero-padded)
`action`	Actions to execute — multiple separated by `,` (format: `EIS=GA=val`)
`repeat`	Repeat days of week — comma-separated (`,1,` = Mon … `,7,` = Sun)

Success reply (timingrsp):

```
{"sn":"08ABEF112233","cmd":"timingrsp","data":{"tid":"123","tname":"My Scene","oR
```

Failure reply:

```
{"sn":"08ABEF112233","cmd":"timingrsp","data":{"error":"1"}}\r\n
```

Appendix: EIS Device Types & Attributes

Device Type	`type`	EIS	Attribute
Standard switch (light)	1	EIS1	On/Off
Dimmable light	1	EIS1	On/Off
		EIS6	Brightness 0–255
		EIS222	Color temperature (2-byte)
Standard roller blind	2	EIS1 / EIS7	Open/Close
Standard split curtain	2	EIS6 (special)	Position % = value/255 × 100
Travel roller blind	2		(same as above)
Travel split curtain	2		
Air conditioning	3	EIS1	On/Off
		EIS5	Set temperature

		EIS60	Room temperature (read)
		EIS61	Mode: 0=Cool, 1=Heat, 2=Fan
		EIS62	Fan speed: 0=Auto, 1=Low, 2=Med, 3=High
Floor heating	4	EIS1	On/Off
		EIS5	Set temperature
		EIS60	Room temperature (read)
Fresh air / ERV	5	EIS1	On/Off
		EIS51	PM2.5
		EIS52	CO2
		EIS53	Fresh air mode control
		EIS54	Humidity control
Scene	6	EIS6	Scene value 0–127
Security / alarm	9	EIS1	Alarm signal (on/off)
		EIS6	Security mode (arm/disarm/bypass…)

Technical Review — Integration into Developer App for 1,000-Unit Residential Project

Protocol Architecture Assessment

Strengths

1. Lean, well-understood stack TCP Socket + JSON over `\r\n` framing is simple to implement on any platform (iOS/Android/backend). No proprietary SDK dependency beyond the KTS host hardware. Straightforward to integrate into Flutter, React Native, or native apps.

2. MD5-based auth with per-session signing `sign = MD5(sn + timestamp + rand + key)` per TCP connection provides basic replay protection. Acceptable for LAN-adjacent use but see security concerns below.

3. Hierarchical zone/object model Zone (`query1`) → Object (`query2`) → Control is a clean abstraction over the underlying KNX group address structure. The `type` + `EIS` taxonomy maps well to UI component rendering.

4. Push-based state updates `push` messages from host to server for real-time state propagation — essential for a multi-user app scenario (1,000 units = potentially concurrent users in same building).

Critical Gaps for 1,000-Unit Scale

1. No multi-tenancy isolation at protocol level

The `sn` field uniquely identifies a **device host** (gateway), not a tenant. In a 1,000-unit building, if each unit has one KTS host, the server must maintain 1,000 persistent TCP connections simultaneously.

Risk: No built-in mechanism to prevent cross-unit access. Authorization (unit owner can only control their `sn`) must be implemented **entirely on the server/app layer** — the protocol itself has zero tenant isolation.

Recommendation: Server-side mapping table: `user_id → allowed_sn[]` . Enforce before forwarding any `control` or `query` command.

---

2. MD5 signature is cryptographically weak

MD5 is broken for security purposes. HMAC-SHA256 would be a drop-in replacement. With MD5:

- The `key` can be brute-forced or rainbow-tabled if traffic is intercepted.
- No forward secrecy — compromised key = all sessions compromised.

Recommendation: Wrap the TCP connection in **TLS 1.2/1.3**. The MD5 `sign` can remain for host identity validation, but all transport must be encrypted. At 1,000 units, a TLS termination proxy (nginx/HAProxy in front of the KTS TCP server) is the practical path.

---

3. No acknowledgment on `push` — state sync reliability unknown

The `push` message has **no ACK** from server to host. If the TCP connection drops mid-push, state updates are lost. The client app will show stale data.

Recommendation: Implement a server-side state cache (Redis or PostgreSQL). On reconnect, server must request a full `query2` + current values for all objects of that `sn` to re-sync state. The app should treat `val:"-1"` as "loading" and trigger a re-query.

---

4. 40KB packet cap with no chunking mechanism

`rsp2` (object query reply) for a feature-rich apartment (lighting + curtains + AC + fresh air + security) could plausibly hit 20–40 objects. At 1,000 units with many objects per zone, the 40KB cap may be hit.

Risk: No pagination or chunking protocol defined. If payload exceeds 40KB, behavior is undefined.

Recommendation: During integration testing, stress-test with maximum object count per zone. If hitting the limit, partition queries by zone (`oRoom` field) rather than querying all at once (`oRoom: "-1"`).

---

5. Heartbeat interval (10–60s) and connection management at scale

1,000 persistent TCP connections with 10–60s heartbeats = significant server load. Standard TCP keepalive at OS level is more efficient.

Recommendation:

- Set heartbeat to 60s (not 10s) at scale.
- Use connection pooling or a message broker (MQTT or WebSocket) for the server↔app leg, reserving the raw TCP only for server↔KTS host.
- Consider an architecture where the **app talks to a REST/WebSocket API server**, and the API server manages persistent TCP connections to all KTS hosts. This decouples the mobile app from raw TCP entirely.

---

6. Scheduled task (`timing`) stored on device — risky for cloud app

The `timing` command pushes schedules **to the KTS host for local execution**. This means:

- Schedules survive server downtime (good).
- But if the host is offline when the schedule fires, behavior is undefined.
- **No central schedule database** — the developer app cannot query existing schedules without the host sending them back.

Recommendation: Maintain a **server-side shadow** of all timing rules. On reconnect (`reg`), sync timing state. The app should only write to the server DB; server pushes to host.

---

7. No error codes beyond `error: 0/1`

Binary error signaling (0=ok, 1=fail) is insufficient for debugging at scale. Network issues, permission failures, invalid EIS commands — all return `error: 1` .

Recommendation: Add error codes in the server middleware layer. Log all raw API exchanges with timestamps for each `sn` . Essential for after-sales support at 1,000 units.

Integration Architecture Recommendation

For a developer app at this scale, **do not expose the KTS TCP protocol directly to mobile clients**. The recommended architecture:

```
Mobile App (iOS/Android)
        ↕  HTTPS / WebSocket (REST or MQTT)
[Developer Backend Server]
  - Auth & tenant isolation (user → sn mapping)
  - State cache (Redis)
  - Schedule database
  - TLS termination
        ↕  TCP Socket (KTS protocol, internal network)
[KTS Host Devices] × 1,000 units
```

This backend abstraction layer resolves all scalability, security, and reliability issues without requiring changes to the KTS firmware or protocol.

Summary Table

Dimension	Assessment	Action Required

Transport security	❌ No TLS on raw TCP	Add TLS termination proxy
Authentication	⚠️ MD5 (weak)	Keep for host ID; enforce TLS at transport
Multi-tenancy	❌ Not in protocol	Implement server-side user→sn mapping
State reliability	⚠️ No ACK on push	Server-side state cache + re-query on reconnect
Scalability (1,000 units)	⚠️ 1,000 persistent TCP connections	Backend proxy layer; heartbeat at 60s
Packet size limits	⚠️ 40KB cap, no chunking	Partition queries by zone
Schedule management	⚠️ Host-side only	Server-side shadow DB
Error granularity	❌ Binary only	Middleware logging layer
Mobile integration	✅ Via backend abstraction	REST/WebSocket API layer
KNX address exposure	ℹ️ Raw GA in protocol	Abstract at backend; never expose GA to app