INTEGRATION GUIDE

Active RFID · F529 · F530

Connect Active RFID Readers to Your Software

F529 and F530 readers with T505, AT507 and AT508 tags: how the reader talks to your server, the frame format byte by byte, tested receivers in Node.js and Python, the reader library, and the Identium RTLS platform API



THIS MANUAL COVERS	VERSION	PLATFORM
Identium Active Reader app (sets the reader's server address)	2.0 and later	Android 7.0 and newer
Active RFID Integration Kit (receivers and reader simulator)	1.0	Node.js 18+ · Python 3.8+
F529 Active Reader SDK (reader library)	1.0 (library 2.0)	Windows · Linux
Identium RTLS platform REST API	4.19	Any HTTP client

Document version 1.0.1 · November 2026

Identium Tech Solutions Pvt. Ltd.
Plot No. 5, First Floor, Raja Dhirsain Marg, Garhi, Sant Nagar, East of Kailash, New Delhi 110065, India
+91 97181 99516 · 011-4714 7839 · WhatsApp +91 82877 32647 · info@identium.in

www.identiumone.com · www.indiarfidstore.com
Creating identities for all

Identium F529 and F530 active RFID readers hear Identium 2.4 GHz active tags – T505 cards, AT507 rugged tags and AT508 asset tags – and send every read to a server over TCP. This guide is for the developer who connects them to software. It explains the conversation between reader and server, the frame format byte by byte with a worked decode, what each tag field means on site and how to turn reads into presence and zones. It then gives three ways to integrate: your own TCP server, with tested receivers in Node.js and Python and a reader simulator to test them; the reader library in the F529 SDK; or the Identium RTLS platform and its REST API.

CONTENTS

01/ How the System Fits Together

02/ The Conversation: Register, Login, Heartbeat, Tag Data

03/ Point the Reader at Your Server

04/ Frame Format

05/ Worked Decode: One Tag-Data Frame

06/ Tag Fields in Practice

07/ From Reads to Presence and Zones

08/ Path A: Your Own TCP Server

09/ Path B: The Reader Library in the F529 SDK

10/ Path C: The Identium RTLS Platform and Its API

11/ Performance and Scaling

12/ Troubleshooting

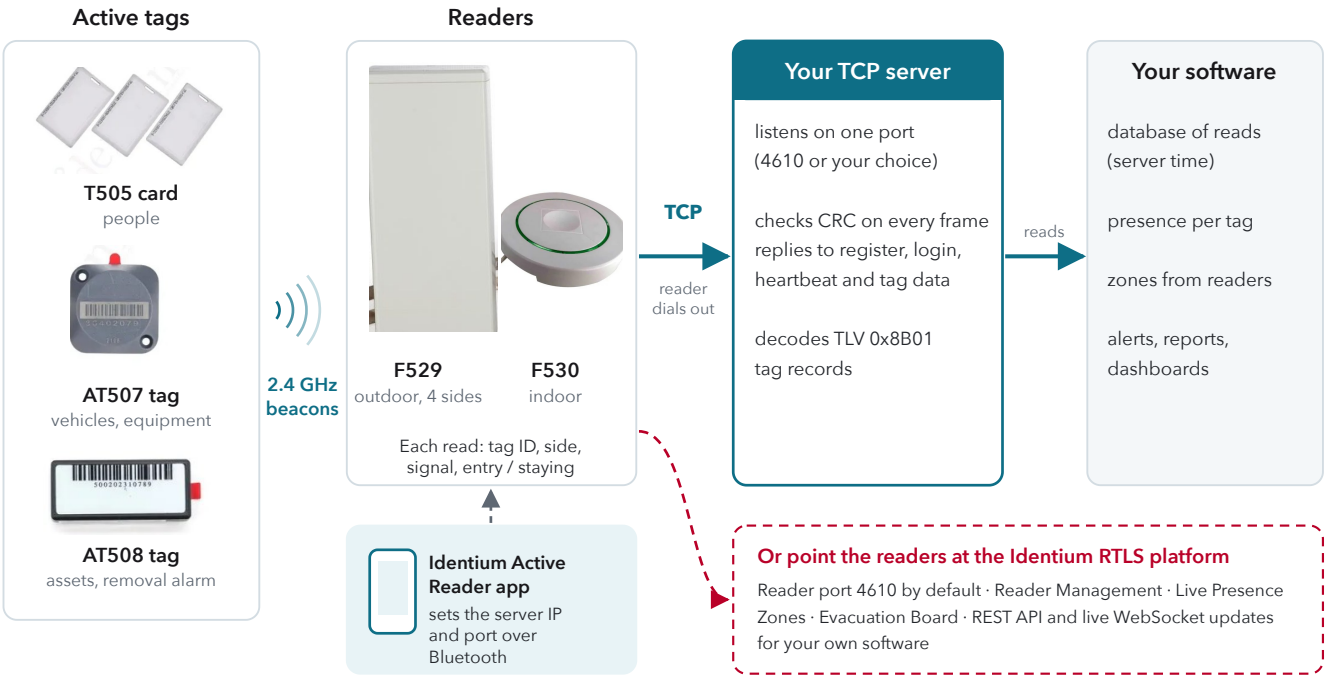
13/ Support

01/ How the System Fits Together

Three things talk to each other. **Tags** beacon on their own on 2.4 GHz. A **reader** hears them and sends each read to a **server** over TCP. The reader always starts the connection: you set your server's IP address and port in the reader once, with the Identium Active Reader app, and from then on the reader dials in, keeps the connection open and reports. Your server only listens and replies.

What travels where

- **Tag → reader:** radio beacons carrying the tag's ID and status. The reader adds the antenna (side), the signal strength (RSSI), entry and staying flags and its own clock time.
- **Reader → your server:** binary frames over one TCP connection, over the reader's mobile data (SIM) or a wired network. Every frame carries the reader's 15-digit **Device ID**, so one listening port can serve many readers.
- **Phone → reader:** Bluetooth, from the Identium Active Reader app, to set the server address and the read settings. The app is a set-up tool; tag reads go to your server.



Each reader opens and keeps one TCP connection to the address and port set in the app; your server listens and replies.

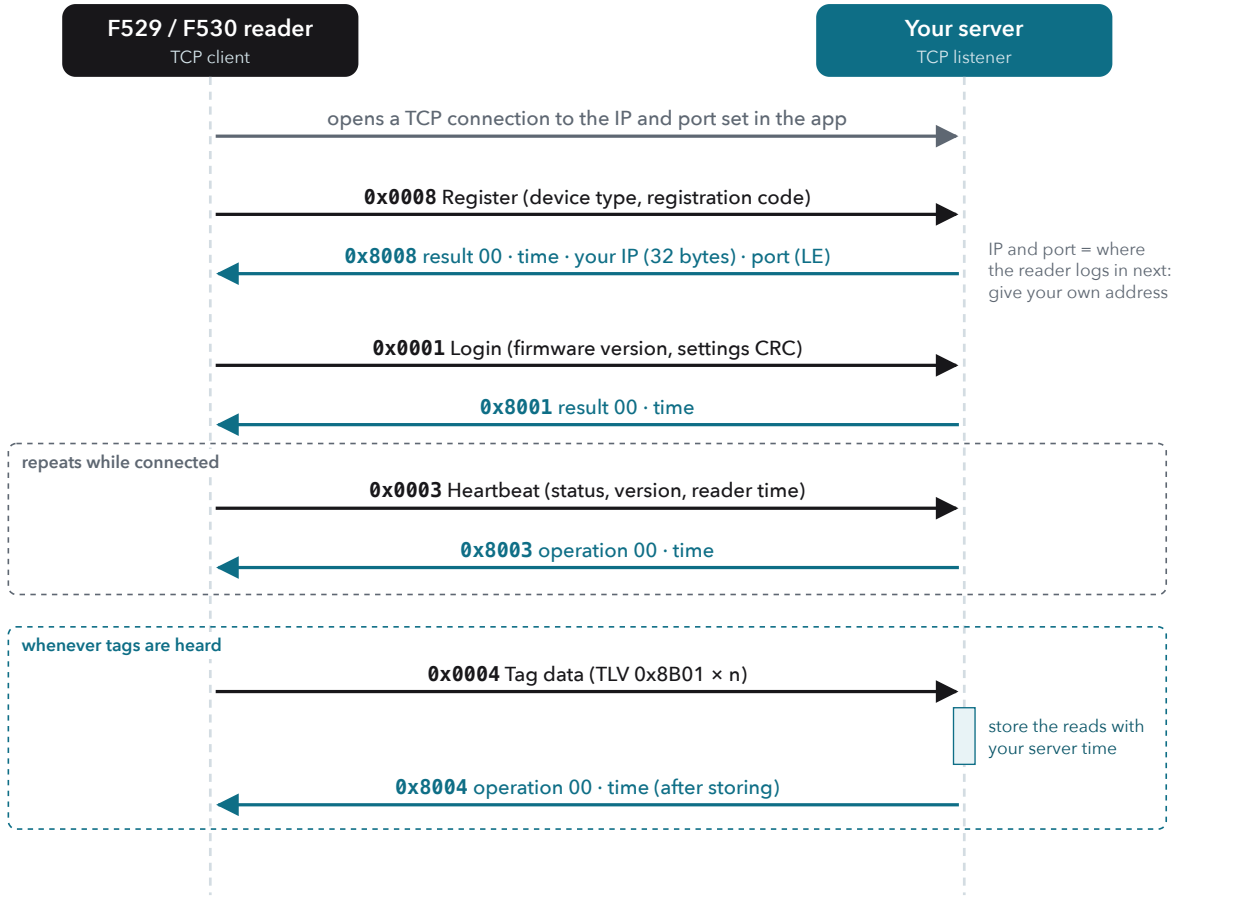
Tags → reader → your TCP server → your software. The dashed path is the alternative: point the readers at the Identium RTLS platform and use its API.

Model	What it is	Use it for
F529	Outdoor active reader. Four built-in directional antennas, one per side, so each read says which side heard the tag. Range up to 400 m depending on tag type, about 100 m typical – confirm by site survey. IP65.	Yards, gates, perimeters, outdoor sites
F530	Indoor active reader in a round housing for wall or ceiling. Up to 200 m in open space with a 7 dBm tag; depends on tag type.	Buildings, entrances, vehicles
T505	Active card, battery-powered, read hands-free. Up to 150 m depending on reader and antenna. Battery rated 3 years.	People: staff, contractors, visitors
AT507	Rugged active tag in a grey ABS housing; fixes with screws or an adhesive pad. Battery rated 3 years.	Vehicles, equipment, outdoor assets
AT508	Small active asset tag with a light sensor that sends an alarm when the tag is removed from the object. Up to 80 m, set in software.	IT assets, instruments, valuables
ACT-F529-EVK	Evaluation kit: 1 × F529, 2 × T505, 2 × AT507.	Your first integration and site trial

Ranges are open-space figures from the product pages; walls, metal, bodies and mounting height shorten them. Buy at www.indiarfidstore.com/ (search by model).

02/ The Conversation: Register, Login, Heartbeat, Tag Data

The reader sends a request and your server answers each one on the same connection with a reply code of request + 0x8000. Four requests are all a receiver needs. The full protocol, including remote configuration, firmware update and tag messages, is in the **Identium F529 Reader-Platform Communication Protocol V1.12** (Resource Hub and SDK `5–Docs/`), sections 5 to 8.



No reply, or a reply the reader cannot accept:
tag data and heartbeats are sent again up to 5 times, then the reader logs in again; a login is tried 10 times, then the reader registers again. A frame that fails the CRC check gets no reply, so the reader resends it.

One connection from start to tag data. The reader starts every exchange.

Reader sends	Reply	Content of your reply
`0x0008` **Register** – device type (2 bytes), registration code (4)	`0x8008`	Result `00` (accepted), your server time (6), the IP address the reader should log in to (32 bytes of text, padded with `00`), the port (2 bytes, **low byte first**). 41 bytes.
`0x0001` **Login** – firmware version (2), settings CRC (2)	`0x8001`	Result `00`, server time. 7 bytes.
`0x0003` **Heartbeat** – work status (2), device status (2), firmware version (2), reader time (6)	`0x8003`	Operation `00` (nothing to do), server time. 7 bytes.
`0x0004` **Tag data** – one TLV per tag read	`0x8004`	Operation `00`, server time. 7 bytes.

Rules your server must follow

- Copy the request's **sequence number**, **protocol version** and **Device ID** into the reply header, set the security flag to `00 00`, and add a correct CRC. A reply that does not match is treated as no reply.
- **The registration reply tells the reader where to log in.** Put in the IP address and port at which the reader reaches the server that should get the data – normally your own server, with the same public IP and port you set in the reader. The reader then logs in there.
- **Reply to tag data only after the reads are stored.** A reader that gets no valid reply sends the same tag data or heartbeat again, up to 5 times, then logs in again; a login is tried 10 times before the reader registers again (protocol sections 7.2–7.4). Storing first and replying second means a crash never loses reads.
- **Drop a frame whose CRC fails and send nothing.** The reader resends it.
- Close a connection that has been silent for a few minutes (the receivers here use 3 minutes) and turn on TCP keepalive: on mobile data a connection can vanish without a close. When a Device ID connects again, treat the new connection as the live one.
- Log any other request code and leave it unanswered; the four above are all a reader needs to report tags.

```
 0  55 AA 00 45 80 08 00 00 00 01 00 01 00 00
14  32 30 32 36 31 31 32 30 30 30 35 32 39 30
28  31 00 00 1A 0B 14 0A 0F 02 32 30 33 2E 30
42  2E 31 31 33 2E 31 30 00 00 00 00 00 00 00
56  00 00 00 00 00 00 00 00 00 00 00 00 00 02
70  12 7B 93
```

Example 0x8008 registration reply (73 bytes): result 00 at byte 30, server time 1A 0B 14 0A 0F 02 = 2026-11-20 10:15:02, IP text "203.0.113.10" from byte 37, port 4610 = 0x1202 sent as 02 12 at bytes 69–70, CRC 7B 93.

03/ Point the Reader at Your Server

You do this once per reader, on site, with the **Identium Active Reader** app on an Android phone over Bluetooth. The **F529 User Manual** (v1.3, Resource Hub) shows every screen of app 2.1 step by step in "Connect the Reader to Your Server"; this guide names the fields the app sets.

Field (tab)	What to do with it
Device ID (Device Info)	Note it. Every frame carries it in bytes 14–29. Register it in your software – or on the platform's Reader Management page – exactly as shown.
Plat IP 1 / Plat Port 1 (Wireless Network)	Your server's public IP address and your reader port, for a reader that sends over mobile data (SIM). Use a numeric IP address.
Plat IP 2 / Plat Port 2 (Wire Network)	The same, for a reader on a wired network. Set the pair that matches the link the reader uses.
Network Status (Device Info)	Shows whether the reader has a link to the server, for example **Wireless connected**.
Total ID in Buffer (Device Info)	Reads waiting inside the reader. A count that keeps rising means the reader cannot reach your server.
Reader clock (Device Info)	The time the reader writes into every read. Set it from the phone's clock.
Antenna Parameters	RSSI cut-off per antenna (read range), Filter Time and Residence Time – see section 06.

1. Start your receiver first (section 08), so you can watch the reader arrive.
2. Connect to the reader in the app and tap **Read Device Info**.
3. Enter your server's IP address and port in the pair for the reader's link, then tap **Set Params**.
4. The reader restarts and the Bluetooth link drops. Wait about 30 seconds, reconnect, tap **Read Device Info** and check **Network Status**. Your receiver logs a connection, then `0x0008` and `0x0001`.

CAUTION

Set Params deletes reads still waiting in the reader. Check **Total ID in Buffer** before you save new server settings, and let waiting reads reach a server first if they matter.

NOTE

The port is your **reader listener** – the port your receiver listens on, such as 4610 – not a website port. Open it for TCP in every firewall on the way: the server's own firewall, the router (port forwarding) and any cloud security group.

Keep the reader port private

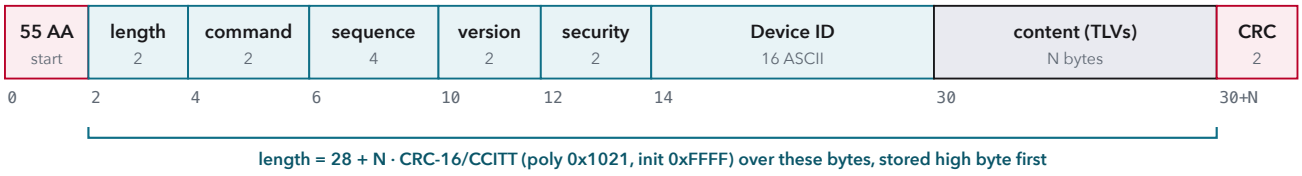
The reader sends no user name or password. Accept reads only from Device IDs you have registered, and where you can, let only your readers reach the port: an allow-list of the readers' addresses, a private mobile network (APN) or a VPN. Frames travel in plain text, the reader's factory setting.

04/ Frame Format

Every frame, in both directions, has the same shape: a start flag, a 28-byte header, the content and a CRC. Numbers are big-endian (high byte first), except the port in the registration reply.

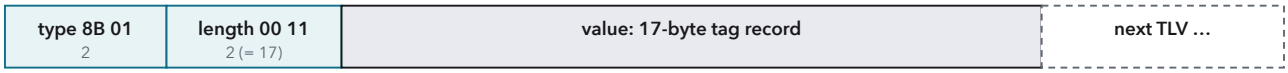
Bytes	Field	Notes
0–1	Start flag	Always `55 AA`. Use it to find the start of a frame in the byte stream.
2–3	Length	28 + content length. It counts from this field to the end of the content, so the whole frame is length + 4 bytes. A value below 28 is not a real frame.
4–5	Command	Requests `0x00xx`; replies `0x80xx`.
6–9	Sequence	Counts up with every request; copy it into your reply. A jump means frames were lost; a reset to a low number means the reader restarted.
10–11	Protocol version	Copy it into your reply.
12–13	Security	Bit 15 = the reader supports encryption, bit 14 = encryption on. Readers send plain text from the factory (`80 00` or `00 00`); reply `00 00`.
14–29	Device ID	The reader's ID as text, padded with `00`.
30 …	Content	Command-specific. Tag data is a list of TLVs (type, length, value).
last 2	CRC	CRC-16/CCITT: polynomial `0x1021`, start value `0xFFFF`, no bit reflection, no final XOR, over bytes 2 to 2 + length − 1. Stored high byte first.

Every frame (both directions)

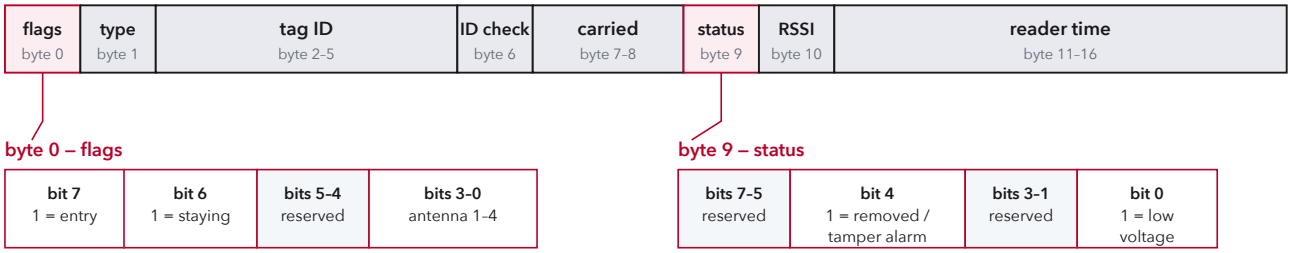


byte offsets shown under each field · all numbers big-endian except the port in the 0x8008 reply

Content of a tag-data frame (0x0004): one TLV per tag read



The 17-byte tag record (TLV 0x8B01)



Frame, TLV and the 17-byte tag record. Offsets are byte numbers from 0.

TIP

Test your CRC function on the example frame in section 05: over its bytes 2 to 71 the result must be **C3 C5**. The same function checks every frame the reader sends and signs every reply.

Tag reads use TLV type `0x8B01` with a 17-byte value (protocol section 8.2). Skip other TLV types by their length; the receivers do this.

05/ Worked Decode: One Tag-Data Frame

This 74-byte frame is what the simulator in the Integration Kit sends (`reader-sim.py --dump`), with the same layout a reader uses: reader `202611200052901`, sequence 4, two tag reads. The receivers, the SDK reader library and the Identium RTLS platform all decode it to the values below.

```
 0  55 AA 00 46 00 04 00 00 00 04
10  00 01 80 00 32 30 32 36 31 31
20  32 30 30 30 35 32 39 30 31 00
30  8B 01 00 11 82 20 30 A1 C4 5E
40  ED 00 00 00 C2 1A 0B 14 0A 0F
50  1E 8B 01 00 11 43 20 5B 0E 77
60  D2 2E 00 00 10 B2 1A 0B 14 0A
70  0F 1E C3 C5
```

Tag-data frame, 10 bytes per row; the number on the left is the offset of the first byte.

Header – bytes 0 to 29

Bytes	Hex	Field	Value
0–1	55 AA	Start flag	
2–3	00 46	Length	70 = 28 header + 42 content; the frame is 70 + 4 = 74 bytes
4–5	00 04	Command	Tag data
6–9	00 00 00 04	Sequence	4
10–11	00 01	Protocol version	0x0001
12–13	80 00	Security	Encryption supported (bit 15), off (bit 14 = 0): plain text
14–29	32 30 … 31 00	Device ID	“202611200052901” and one `00`

Two tag records – bytes 30 to 71

Bytes	Field	Record 1	Record 2
30–31 · 51–52	TLV type	`8B 01` active tag	`8B 01`
32–33 · 53–54	TLV length	`00 11` = 17	`00 11` = 17
34 · 55	Flags	`82` = 1000 0010: entry 1, staying 0, antenna 2 (south)	`43` = 0100 0011: entry 0, staying 1 (so the entry bit is not valid), antenna 3 (west)
35 · 56	Tag type	`20`	`20`
36–39 · 57–60	Tag ID	`30 A1 C4 5E` → **30A1C45E**	`5B 0E 77 D2` → **5B0E77D2**
40 · 61	ID check	`ED`: 20+30+A1+C4+5E = 0x213; 0x100 − 0x13 = ED ✓	`2E`: 20+5B+0E+77+D2 = 0x1D2; 0x100 − 0xD2 = 2E ✓
41–42 · 62–63	Carried info	`00 00`	`00 00`
43 · 64	Status	`00` normal	`10` = bit 4: removed / tamper alarm
44 · 65	RSSI	`C2` = 194 − 256 = **−62 dBm**	`B2` = 178 − 256 = **−78 dBm**
45–50 · 66–71	Reader time	`1A 0B 14 0A 0F 1E` = 2026-11-20 10:15:30	same

Bytes 72–73, `C3 C5`, are the CRC over bytes 2 to 71. In plain words: a tag **30A1C45E** came into the reader's area on its south side at −62 dBm, and tag **5B0E77D2** is staying on the west side at −78 dBm and reports a removal alarm.

```
{"received": "2026-11-20T04:45:31+00:00", "device": "202611200052901",
 "tag_id": "30A1C45E", "antenna": 2, "entry": 1, "staying": 0, "tag_type": 32,
 "rssi": -62, "low_voltage": 0, "tamper": 0, "reader_time": "2026-11-20 10:15:30"}
{"received": "2026-11-20T04:45:31+00:00", "device": "202611200052901",
 "tag_id": "5B0E77D2", "antenna": 3, "entry": 0, "staying": 1, "tag_type": 32,
 "rssi": -78, "low_voltage": 0, "tamper": 1, "reader_time": "2026-11-20 10:15:30"}
```

What the receivers print for this frame (one JSON line per read, wrapped here). `received` is your server's clock in UTC; `reader_time` is the reader's own clock, here India time.

Your reply

```
  0  55 AA 00 23 80 04 00 00 00 04 00 01 00
 13  00 32 30 32 36 31 31 32 30 30 30 35 32
 26  39 30 31 00 00 1A 0B 14 0A 0F 1F 67 5C
```

The 39-byte 0x8004 reply: length `00 23` = 28 + 7, command `80 04`, sequence `00 00 00 04` and version `00 01` copied, security `00 00`, Device ID copied, operation `00`, server time `1A 0B 14 0A 0F 1F` = 2026-11-20 10:15:31, CRC `67 5C`.

06/ Tag Fields in Practice

Field	What it means	How to use it
Tag ID	4 bytes, shown as 8 hex characters.	Your key for the tag. Some labels print the ID as a number instead: the SDK reader library converts it (`RFID_TagidConvert`) to a 10-digit decimal or a 12-digit structured form – 30A1C45E becomes 0815907934 or 120501130094. Match whichever is on your tags, and store the hex form.
Antenna	On the F529: 1 = east, 2 = south, 3 = west, 4 = north side of the reader.	Tells you which side heard the tag. Write down how each reader is mounted (which side faces the gate, the yard …) at installation, or the numbers mean nothing later.
Entry	1 = the tag came into the reader's area; 0 = it left.	An arrival or departure event. Ignore it when **staying** is 1.
Staying	1 = the tag is staying in the reader's area.	The tag is still there; the entry bit is not valid in this record.
Tag type	A code for the tag family (`20` in the example).	Store it; identify tags by ID.
RSSI	Signal strength, signed dBm. Nearer 0 = stronger.	Relative only: distance, tag orientation, bodies and metal all change it. Compare reads of the same tag between readers and over time.
Status	Bit 4 = removed / tamper alarm; bit 0 = low voltage (protocol section 8.2).	The AT508's light sensor sends an alarm when the tag is removed from its object. Test with your own tags which bits they set before you build alerts on them.
Carried info	2 bytes some tag types use for extra data, such as a sensor value.	Store it as is.
Reader time	The reader's clock when it heard the tag.	Use **your server's receive time** as the event time and keep the reader time for audit: a reader clock may be unset, and reads that waited in the buffer arrive late.

How often a tag is reported

Two settings in the app decide this. **Filter Time** – the same tag is reported only once within this time (180 s by default). **Residence Time** – how long the reader keeps remembering a tag after it stops hearing it (0, or 60 s and more; 180 s by default). The reader reports a tag when it arrives; in Identium's own testing a tag that then stays in front of a reader can go quiet over TCP for long stretches. So a missing read does not by itself mean a missing tag: size your presence windows to these settings, use the entry and staying flags, and test on site with the settings you will run.

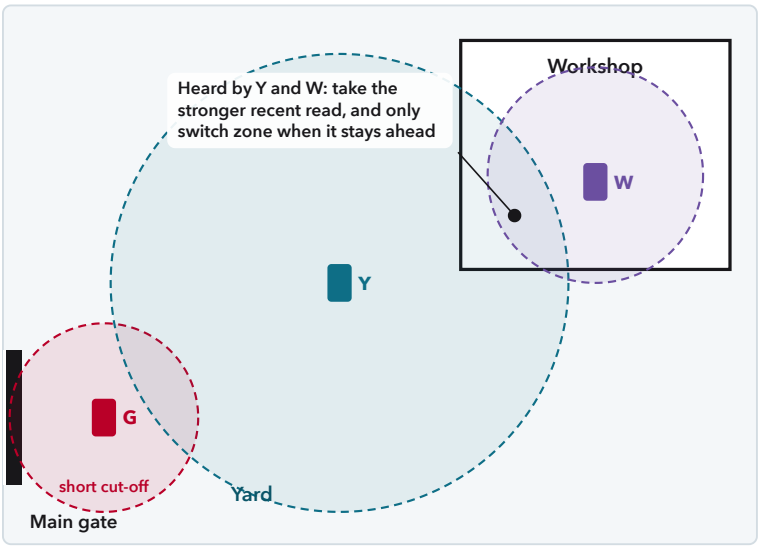
Read range

Each antenna has an **RSSI cut-off** on the Antenna Parameters tab: the reader ignores tags weaker than it. More negative = longer range, nearer 0 = shorter (default −88 dBm). Set range at the reader first, then apply your own RSSI threshold in software where zones need a sharper edge.

07/ From Reads to Presence and Zones

Active RFID tells you which reader heard a tag, on which side (F529), how strongly and when. That gives **zone-level** presence – "in the workshop", "at the gate" – rather than a position on a map.

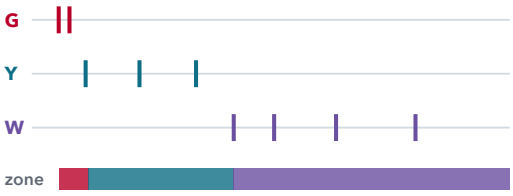
Site plan: zones are groups of readers



Zone	Readers	Window
Gate	G (F529)	30 s
Yard	Y (F529)	5 min
Workshop	W (F530)	5 min

Windows are examples; tune them on site.

One tag over ten minutes



Ticks = reads; the bar = the zone you report. A zone is left when its window runs out, or another zone takes over.

The answer is "near reader Y, on its east side, 40 s ago" – a zone, not an x/y position. For positions on a map, use the Identium BLE AoA system. Readers and zones above are an example layout; confirm every read area with a site survey.

Zones are groups of readers. Overlaps are normal; decide them with recent signal strength and a little hysteresis.

1. **Map readers to zones.** A zone is one or more readers (on the F529, optionally one side of a reader). A big area can use several readers; a gate can use one reader with a short cut-off.
2. **Give each zone a presence window.** A tag is in a zone while its last read from one of the zone's readers is newer than the window. The Identium RTLS platform uses 300 s per zone by default (minimum 5 s) and shows a tag as present under 60 s, stale under 10 minutes and missing after that. Use wider windows where tags stay still and narrower ones at gates.
3. **Filter weak reads.** Ignore reads below an RSSI threshold at zone edges, on top of the reader's cut-off.
4. **Resolve overlaps.** When two zones hear the same tag, take the zone with the stronger recent reads (for example the average of the last three), and move the tag only when the new zone stays ahead for a few reads, so a tag on a boundary does not flicker.
5. **Use events as well as windows.** A record with entry 0 and staying 0 is a departure; an arrival at another zone also ends the old one. Let the window catch the rest.
6. **For direction at a gate,** place one reader outside and one inside, both with short cut-offs: the order in which they hear a tag gives in or out.

> **NOTE**
> For roll calls and mustering, keep windows generous so that a person who steps just out of range is not dropped from the list. For positions on a floor plan, use the Identium BLE AoA system with the Identium Location Engine.

08/ Path A: Your Own TCP Server

Download the **Active RFID Integration Kit** (`Identium-Active-RFID-Integration-Kit-v1.0.zip`) from **www.identiumone.com/resourcehub/**. Nothing to install beyond Node.js or Python.

File	What it does
`receivers/node/receiver.js`	TCP receiver for Node.js 18 or later. No packages.
`receivers/python/receiver.py`	The same receiver for Python 3.8 or later. Standard library only.
`simulator/reader-sim.py`	Plays the part of a reader against your receiver and checks every reply: start flag, length, CRC, reply code, copied sequence, version and Device ID, result byte.
`frames/example-frames.txt`	The example register, login, heartbeat and tag-data frames as hex.

What every receiver must do

- Listen on one TCP port; keep each reader's connection open. One port serves many readers.
- Cut frames out of each connection's byte stream: find `55 AA`, read the length, wait for length + 4 bytes – TCP may split a frame or join two. If the length is not plausible, look for the next `55 AA`.
- Check the CRC; on failure drop the frame and send nothing.
- Decode TLV `0x8B01` records, store each read with your receive time and the Device ID, then answer as in section 02. Use TCP keepalive and close connections silent for a few minutes.

Test it in five minutes

1. Start a receiver: `node receivers/node/receiver.js --port 4610 --ip 127.0.0.1` or `python3 receivers/python/receiver.py --port 4610 --ip 127.0.0.1`. The `--ip` value goes back to the reader in the registration reply: on a real server, give the address the reader uses to reach it.
2. In a second window run `python3 simulator/reader-sim.py --port 4610`. Every line should say PASS, and the receiver prints the two reads of section 05 as JSON.
3. Run the harder framing test: `python3 simulator/reader-sim.py --port 4610 --chunk 7 --noise --together --bad-crc` (frames in small pieces, stray bytes, two frames at once, one broken CRC).
4. Replace the line that prints each read (`console.log` or `print`) with a write to your database, and send the `0x8004` reply only after the write succeeds. Test your own server with the simulator before you point a reader at it; `--loop 10` keeps it reporting every 10 seconds.

```
Connected to 127.0.0.1:4610 as reader 202611200052901
  PASS  register 0x0008 -> 0x8008, 73 bytes, CRC ok, server time 2026-11-20 18:35:29
  PASS  registration reply points the reader to 127.0.0.1:4610
  PASS  login 0x0001 -> 0x8001, 39 bytes, CRC ok, server time 2026-11-20 18:35:29
  PASS  heartbeat 0x0003 -> 0x8003, 39 bytes, CRC ok, server time 2026-11-20 18:35:29
  PASS  tag data with a broken CRC: no reply (the reader would resend)
  PASS  tag data 0x0004 (2 tags) -> 0x8004, 39 bytes, CRC ok, server time 2026-11-20 18:35:31
  PASS  heartbeat sent together with tag data -> 0x8003, 39 bytes, CRC ok, ...
RESULT: PASS
```

Simulator output for the harder test against the Node.js receiver.

Node.js receiver — receivers/node/receiver.js (part 1 of 2)

```
// Identium active RFID receiver — Node.js 18 or later, no packages needed.
// Run:  node receiver.js --port 4610 --ip 203.0.113.10   (reads: stdout as JSON lines)
// --ip: the address the reader uses to reach this server (sent back in the registration reply).
'use strict';
const net = require('net');
const arg = (k, d) => (i => (i > 0 ? process.argv[i + 1] : d))(process.argv.indexOf('--' + k));
const PORT = Number(arg('port', 4610)), MY_IP = arg('ip', '127.0.0.1');
const IDLE_MS = 180000;                       // close a socket that is silent for 3 minutes
const log = (...a) => console.error(new Date().toTimeString().slice(0, 8), ...a);
const p2 = n => String(n).padStart(2, '0');

function crc16(buf) {                         // CRC-16/CCITT: poly 0x1021, init 0xFFFF
  let crc = 0xFFFF;
  for (const b of buf) {
    crc ^= b << 8;
    for (let i = 0; i < 8; i++) crc = (crc & 0x8000 ? (crc << 1) ^ 0x1021 : crc << 1) & 0xFFFF;
  }
  return crc;
}
function now6() {                             // year-2000, month, day, hour, minute, second
  const t = new Date();
  return Buffer.from([t.getFullYear() - 2000, t.getMonth() + 1, t.getDate(),
                      t.getHours(), t.getMinutes(), t.getSeconds()]);
}
function reply(req, cmd, content) {           // mirror sequence, version, Device ID; security 0
  const f = Buffer.alloc(32 + content.length);
  f[0] = 0x55; f[1] = 0xAA;
  f.writeUInt16BE(28 + content.length, 2);
  f.writeUInt16BE(cmd, 4);
  req.copy(f, 6, 6, 12);                      // sequence (4) + protocol version (2)
  req.copy(f, 14, 14, 30);                    // Device ID (16)
  content.copy(f, 30);
  f.writeUInt16BE(crc16(f.subarray(2, f.length - 2)), f.length - 2);
  return f;
}
function tags(f, total) {                     // TLV 0x8B01 records, 17 bytes each
  const out = [], end = 2 + total;
  for (let i = 30; i + 4 <= end;) {
    const type = f.readUInt16BE(i), len = f.readUInt16BE(i + 2);
    const v = f.subarray(i + 4, i + 4 + len);
    i += 4 + len;
    if (i > end) break;
    if (type !== 0x8B01 || len !== 17) continue;
    out.push({
      tag_id: v.subarray(2, 6).toString('hex').toUpperCase(), antenna: v[0] & 0x0F,
      entry: v[0] >> 7 & 1, staying: v[0] >> 6 & 1, tag_type: v[1], rssi: v.readInt8(10),
      low_voltage: v[9] & 1, tamper: v[9] >> 4 & 1,
      reader_time: `${2000 + v[11]}-${p2(v[12])}-${p2(v[13])} `
                 + `${p2(v[14])}:${p2(v[15])}:${p2(v[16])}`,
    });
  }
  return out;
}
```

Node.js receiver (part 2 of 2)

```
function handle(sock, f, total) {
  const cmd = f.readUInt16BE(4), dev = f.toString('latin1', 14, 30).split('\0')[0];
  const ok = Buffer.concat([Buffer.from([0x00]), now6()]);  // result/operation 0 + server time
  if (cmd === 0x0008) {                       // registration: tell the reader where to log in
    const ip = Buffer.alloc(32);
    ip.write(MY_IP.slice(0, 31), 'ascii');
    const port = Buffer.from([PORT & 0xFF, PORT >> 8]);     // low byte first
    sock.write(reply(f, 0x8008, Buffer.concat([ok, ip, port])));
  } else if (cmd === 0x0001 || cmd === 0x0003) {            // login, heartbeat
    sock.write(reply(f, cmd | 0x8000, ok));
  } else if (cmd === 0x0004) {                // tag data: store the reads first, then reply
    const received = new Date().toISOString();
    for (const t of tags(f, total)) console.log(JSON.stringify({ received, device: dev, ...t }));
    sock.write(reply(f, 0x8004, ok));
  }
  log(dev, `cmd 0x${cmd.toString(16).padStart(4, '0')} seq ${f.readUInt32BE(6)}`);
}

net.createServer(sock => {
  const peer = `${sock.remoteAddress}:${sock.remotePort}`;
  let buf = Buffer.alloc(0);
  log('connected', peer);
  sock.setKeepAlive(true, 30000);
  sock.setTimeout(IDLE_MS, () => sock.destroy());
  sock.on('error', e => log(peer, e.message));
  sock.on('close', () => log('closed', peer));
  sock.on('data', chunk => {
    buf = Buffer.concat([buf, chunk]);
    for (;;) {
      const s = buf.indexOf(Buffer.from([0x55, 0xAA]));
      if (s < 0) {                            // no start flag yet: keep a trailing 0x55 only
        buf = buf.subarray(buf[buf.length - 1] === 0x55 ? buf.length - 1 : buf.length);
        break;
      }
      buf = buf.subarray(s);
      if (buf.length < 4) break;
      const total = buf.readUInt16BE(2);
      if (total < 28 || total > 4096) { buf = buf.subarray(2); continue; } // not a real frame
      if (buf.length < total + 4) break;      // wait for the rest of the frame
      const f = buf.subarray(0, total + 4);
      buf = buf.subarray(total + 4);
      if (crc16(f.subarray(2, 2 + total)) !== f.readUInt16BE(2 + total)) {
        log(peer, 'CRC error - frame dropped, no reply');  // the reader sends it again
        continue;
      }
      handle(sock, f, total);
    }
  });
}).listen(PORT, () => log('listening on port', PORT, '- readers are told to use', MY_IP));
```

Python receiver — receivers/python/receiver.py (part 1 of 2)

```
#!/usr/bin/env python3
"""Identium active RFID receiver - Python 3.8 or later, standard library only.
Run:  python3 receiver.py --port 4610 --ip 203.0.113.10
--ip: the address the reader uses to reach this server (sent back in the registration reply).
Tag reads go to stdout as JSON lines; connection events go to stderr."""
import argparse, asyncio, datetime, json, socket, sys

ap = argparse.ArgumentParser()
ap.add_argument('--port', type=int, default=4610)
ap.add_argument('--ip', default='127.0.0.1')
A = ap.parse_args()
IDLE = 180                                  # close a socket that is silent for 3 minutes

def log(*a):
    print(datetime.datetime.now().strftime('%H:%M:%S'), *a, file=sys.stderr, flush=True)

def crc16(data):                            # CRC-16/CCITT: poly 0x1021, init 0xFFFF
    crc = 0xFFFF
    for b in data:
        crc ^= b << 8
        for _ in range(8):
            crc = ((crc << 1) ^ 0x1021 if crc & 0x8000 else crc << 1) & 0xFFFF
    return crc

def now6():                                 # year-2000, month, day, hour, minute, second
    t = datetime.datetime.now()
    return bytes([t.year - 2000, t.month, t.day, t.hour, t.minute, t.second])

def reply(req, cmd, content):               # mirror sequence, version, Device ID; security 0
    body = ((28 + len(content)).to_bytes(2, 'big') + cmd.to_bytes(2, 'big')
            + req[6:12] + b'\0\0' + req[14:30] + content)
    return b'\x55\xaa' + body + crc16(body).to_bytes(2, 'big')

def tags(f, total):                         # TLV 0x8B01 records, 17 bytes each
    out, i, end = [], 30, 2 + total
    while i + 4 <= end:
        typ, n = int.from_bytes(f[i:i+2], 'big'), int.from_bytes(f[i+2:i+4], 'big')
        v, i = f[i+4:i+4+n], i + 4 + n
        if i > end: break
        if typ != 0x8B01 or n != 17: continue
        out.append({'tag_id': v[2:6].hex().upper(), 'antenna': v[0] & 0x0F,
                    'entry': v[0] >> 7 & 1, 'staying': v[0] >> 6 & 1, 'tag_type': v[1],
                    'rssi': v[10] - 256 if v[10] > 127 else v[10],
                    'low_voltage': v[9] & 1, 'tamper': v[9] >> 4 & 1,
                    'reader_time': '20%02d-%02d-%02d %02d:%02d:%02d' % tuple(v[11:17])})
    return out
```

Python receiver (part 2 of 2)

```
def handle(f, total, w):
    cmd = int.from_bytes(f[4:6], 'big')
    dev = f[14:30].split(b'\0')[0].decode('ascii', 'replace')
    ok = b'\x00' + now6()                  # result/operation 0 + server time
    if cmd == 0x0008:                      # registration: tell the reader where to log in
        ip = A.ip.encode()[:31].ljust(32, b'\0')
        w.write(reply(f, 0x8008, ok + ip + A.port.to_bytes(2, 'little')))
    elif cmd in (0x0001, 0x0003):          # login, heartbeat
        w.write(reply(f, cmd | 0x8000, ok))
    elif cmd == 0x0004:                    # tag data: store the reads first, then reply
        received = datetime.datetime.now(datetime.timezone.utc).isoformat(timespec='seconds')
        for t in tags(f, total):
            print(json.dumps({'received': received, 'device': dev, **t}), flush=True)
        w.write(reply(f, 0x8004, ok))
    log(dev, 'cmd 0x%04x seq %d' % (cmd, int.from_bytes(f[6:10], 'big')))

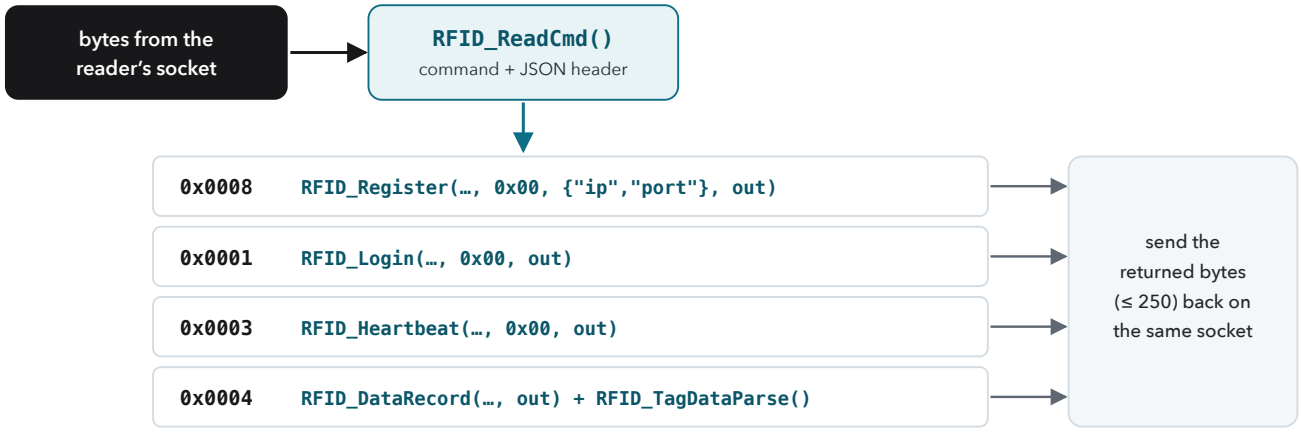
async def serve(r, w):
    peer = w.get_extra_info('peername')
    w.get_extra_info('socket').setsockopt(socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)
    log('connected', peer)
    buf = b''
    try:
        while True:
            chunk = await asyncio.wait_for(r.read(4096), IDLE)
            if not chunk: break
            buf += chunk
            while True:
                s = buf.find(b'\x55\xaa')
                if s < 0:                  # no start flag yet: keep a trailing 0x55 only
                    buf = buf[-1:] if buf[-1:] == b'\x55' else b''
                    break
                buf = buf[s:]
                if len(buf) < 4: break
                total = int.from_bytes(buf[2:4], 'big')
                if not 28 <= total <= 4096: # not a real frame: look for the next 55 AA
                    buf = buf[2:]; continue
                if len(buf) < total + 4: break  # wait for the rest of the frame
                f, buf = buf[:total + 4], buf[total + 4:]
                if crc16(f[2:2 + total]) != int.from_bytes(f[2 + total:], 'big'):
                    log(peer, 'CRC error – frame dropped, no reply')  # the reader resends
                    continue
                handle(f, total, w)
            await w.drain()
    except (asyncio.TimeoutError, ConnectionError):
        pass
    finally:
        log('closed', peer); w.close()

async def main():
    srv = await asyncio.start_server(serve, '0.0.0.0', A.port)
    log('listening on port', A.port, '– readers are told to use', A.ip)
    async with srv:
        await srv.serve_forever()

asyncio.run(main())
```

09/ Path B: The Reader Library in the F529 SDK

The **F529 Active Reader SDK** (**Active Reader – SDK & Demo**, `Identium-F529-Active-Reader-SDK-1.0.zip`, Resource Hub) contains a C reader library that checks frames, builds every reply and turns tag records into JSON. It is prebuilt for Windows (32- and 64-bit) and builds on Linux with `make`; the package has a Java (JNA) sample, the C++ source of a Windows server demo and the protocol documents. The function reference is the **Identium F529 Reader Library API Guide**; the **F529 User Manual** ("For Developers: SDK Quick Start") shows how to run the Windows demo against a reader.



Negative return values: −101 parameter error · −102 CRC check failed (send nothing) · −103 JSON error
RFID_TagDataParse() returns {"TAG":[…]} with tagid, antenna, intensity, entry, staying, alarm, status, time

Call order for every frame the reader sends.

1. Cut the byte stream into single frames yourself (find `55 AA`, read the length, wait for length + 4 bytes), exactly as the receivers in section 08 do. **Pass the library one complete frame per call:** it checks the CRC over the whole buffer you give it, so two frames joined together, or half a frame, return −102.
2. Call `RFID_ReadCmd()`. It returns the command and a JSON header with `device`, `seqno` and `version`.
3. Build the reply: `0x0008` → `RFID_Register(frame, len, 0x00, "{\"ip\":\"203.0.113.10\",\"port\":4610}", out)` with the address the reader should log in to; `0x0001` → `RFID_Login()`; `0x0003` → `RFID_Heartbeat()`; `0x0004` → `RFID_DataRecord()`. Each returns the reply length (at most 250 bytes) to send back.
4. For `0x0004` also call `RFID_TagDataParse()` (give it a buffer of at least 10 KB). It returns `{"TAG":[…]}` with `tagid`, `tagtype`, `tagdata`, `antenna`, `intensity` (RSSI), `entry`, `staying`, `alarm`, `status` and `time`. `alarm` is the status byte masked: 16 = removed / tamper, 1 = low voltage.
5. Treat negative results as errors: −101 parameter error, −102 invalid frame (CRC), −103 JSON error. On −102 send nothing; the reader resends.

> **NOTE**
> Checked for this guide: the library built from the SDK source decodes the section 05 frame to the same values as the receivers, and its replies match the receivers' replies byte for byte apart from the server time, the CRC over it and the copied security flag.

10/ Path C: The Identium RTLS Platform and Its API

The Identium RTLS platform receives the readers for you and gives you web pages and an API. Use it when you want live tracking, presence, zones and an evacuation board without writing the TCP side.

1. In the Identium Active Reader app, set the reader's server IP to the platform's address and the port to the platform's reader port – **4610** unless Identium has given you another one. A reader can also have its own port, set on Reader Management.
2. On **Reader Management** add the reader with its **Device ID** exactly as the app shows it, a name and a zone label. The status turns **online** when its frames arrive.
3. On **Tag Registration** give each tag ID a name, an asset type (Personnel, Equipment, Vehicle, Asset or Other) and an owner. Tags not yet registered still appear on **Raw Inventory**.
4. On **Zones** create zones from readers, with a zone type and a presence window. **Live Presence**, **Live Tracking** and the **Evacuation Board** then work from the reads.

REST API

Sign in once with a user created for your integration, keep the session cookie (valid 24 hours) and send it with every call. Replies are JSON with `"success": true` or `"success": false` and a `message`. If your platform is served under a path prefix, put the prefix before `/api`.

Call	What it does
`POST /api/auth/login`	Body `{"username", "password"}`. Sets the session cookie; 401 on a wrong password.
`GET /api/readers`	Readers with `device_id`, `name`, `zone`, `ip_address`, `tcp_port`, `status`, `last_seen`.
`POST /api/readers`	Add a reader: `device_id` and `name` required; `zone`, `tcp_port`, `notes` optional.
`GET /api/tags` · `POST /api/tags`	List tags; add one with `tag_id` (stored in capitals), `name`, `asset_type`, `owner`, `description`.

Call	What it does
`GET /api/detections/recent`	Latest read per reader and tag with `count`, `first_seen`, `last_seen`, `antenna`, `intensity`. Options: `limit`, `window` (seconds), `include_unregistered=1`, `grouped=0` for raw rows.
`GET /api/detections/all`	Paged reads (`limit` up to 500, `offset`, `search`), registered and unregistered tags, with a `total`.
`GET /api/detections/stats`	Active tags, online readers, reads today, reads per reader, and per-reader link figures: frames received, lost frames, CRC errors, reconnects.
`GET /api/presence`	Every registered tag with its last read, reader, antenna, RSSI and `presence` (present, stale or missing), plus a tally.
`GET /api/zones/live`	Active zones with their readers and occupants; evacuation and muster zones also list registered personnel not yet seen there.
`POST /api/zones` · `PUT /api/zones/:id/readers`	Create a zone; set its readers with `{"readers": [device IDs]}`. Admin or manager users only.
`GET /api/analytics/overview?hours=24`	Reads over time, per reader, per antenna, RSSI histogram, top tags with dwell time. `hours`: 1, 6, 24, 72, 168, 720 or 2160.
`GET /api/analytics/tag/:tagId?hours=24`	One tag's summary, time per zone and timeline. Give the tag ID in capitals.

```
BASE=https://your-platform-address            # the address Identium gives you
curl -c jar.txt -H "Content-Type: application/json" \
     -d '{"username":"integration","password":"<your password>"}' $BASE/api/auth/login
curl -b jar.txt "$BASE/api/presence"
curl -b jar.txt "$BASE/api/detections/recent?window=300&include_unregistered=1"
curl -b jar.txt -H "Content-Type: application/json" \
     -d '{"tag_id":"30A1C45E","name":"Visitor card 12","asset_type":"Personnel"}' $BASE/api/tags
```

Sign in, then read presence and recent reads, and register a tag.

Live updates

Open a WebSocket to the platform with the same session cookie. Each message is `{"type": …, "data": …}`: `tag_update` carries `device_id`, `reader_name`, `reader_zone` and the `tags` just read; `reader_status` carries `device_id`, `status` (online or offline) and `last_seen`; `stats_update` carries the live counters.

11 / Performance and Scaling

- **One connection per reader.** Readers keep their TCP connection open, so a server holds one socket per reader. One listening port serves them all; the Device ID in each frame tells them apart. The Identium RTLS platform can also give a reader its own port.
- **Every read is a row.** Index reads by Device ID, tag ID and time, insert each frame's reads in one batch, and plan a retention policy from the start.

- **Reads can arrive late and in bursts.** While a reader cannot reach your server it keeps reads in its buffer (Total ID in Buffer) and sends them once the link is back. Stamp them with your receive time, keep the reader time, and do not raise "missing" alarms during a reader outage.
- **Measure the link with sequence numbers.** A jump in a reader's sequence counts lost frames; count CRC failures and reconnects per reader too. The platform reports all three in `/api/detections/stats`.
- **Mobile data costs money.** Choose each SIM's data plan to suit the number of tags and how often they are reported (Filter Time).
- **Expect overlap.** Several readers hearing one tag is normal and is what zone logic resolves (section 07).

12/ Troubleshooting

Connection

Problem	What to check
Your receiver never logs a connection	The reader's server IP and port (the pair that matches its link); a numeric public IP for a reader on mobile data; the port open in the server firewall, router and cloud security group. Test the port from another network with `nc -vz <ip> <port>`. In the app, **Network Status** shows the link and a rising **Total ID in Buffer** confirms the reader cannot reach you.
Connects, registers, then disconnects or starts over	Your replies: wrong CRC, sequence or Device ID not copied, wrong reply code. Run the simulator against your server – it names the failing check.
Registers but never logs in to you	The IP and port in your registration reply. The reader logs in where that reply points; it must be an address the reader can reach (your public IP), not a private or loopback address.
CRC errors on every frame	Compute over bytes 2 to 2 + length − 1 (not the `55 AA`), start `0xFFFF`, polynomial `0x1021`, no reflection; the example frame must give `C3 C5`. Cut frames by the length field before checking.
Occasional CRC errors	Normal on a weak mobile link. Drop the frame without a reply; the reader resends.
Two readers keep replacing each other	Both report the same Device ID. Check **Device ID** on Device Info for each unit.

Tag data

Problem	What to check
Heartbeats arrive, tag data does not	No tag in range or awake; an RSSI cut-off too close to 0 on **Antenna Parameters**; the reader left in antenna test mode (**Antenna Debug** – check **Reader mode** on Antenna Info).
The same reads arrive twice	Your reply came late or was invalid, so the reader resent. Reply promptly and ignore a frame whose Device ID and sequence number you have already stored. Raise **Filter Time** to cut repeat reports.
A tag shows once, then looks gone while it is still there	A tag that stays put is reported rarely. Widen the presence window, use the staying flag, and test your Filter Time and Residence Time on site (section 06).
Read times are wrong or jump	The reader clock is not set, or buffered reads arrived late. Use your receive time; set the reader clock from the phone in the app.
Wrong side reported	Antennas are numbered east, south, west, north relative to the F529 as mounted. Record the orientation and check each side with **Antenna Debug**.
Tag ID does not match the label	The label may show the decimal or 12-digit structured form; convert with `RFID_TagidConvert()` (section 06).

13/ Support

For help with installation, integration or a fault that the steps above do not clear, contact Identium. Please keep the product model, the software version shown in the app and a short description (or screenshot) of the problem ready.

Company	Identium Tech Solutions Pvt. Ltd.
Address	Plot No. 5, First Floor, Raja Dhirsain Marg, Garhi, Sant Nagar, East of Kailash, New Delhi 110065, India
Phone	+91 97181 99516 · 011-4714 7839
WhatsApp	+91 82877 32647
Email	info@identium.in
Software and documents	www.identiumone.com/resourcehub/
Buy hardware and tags	www.indiarfidstore.com

This product and its software are updated from time to time; screens and specifications may change without notice. Check the Resource Hub for the latest version of this manual.