



RTKdata PPK API (v1)

Reference-station data for post-processing: send the coordinate you flew at and a time window, get RINEX back.

VERSION

v1 (2026-07-15, revised 2026-10-01)

AUDIENCE

technical integration team

SUPPORT

jonas@rtkdata.com

CONTENTS

1	Start here	3
2	Base URL and conventions	4
	Every response is HTTP 200, including errors	4
	Times are UTC, and both hour bounds are inclusive	4
	Migrating from another PPK service	5
3	Authentication	5
	POST /user/signin	5
	GET /user/info	6
4	Ordering data	6
	POST /user/download	6
	How we choose the station, and what we check	8
	Read distanceKm before you process	9
5	Stations	9
	GET /user/station	9
	GET /user/station/{code}	10
	Pick a frame, and pick its epoch with it	11
	GET /user/station/list	11
6	Polling and the archive	11
	GET /download/status/{orderId}	11
	GET /download/{orderId}	13
	What is in the archive	13
	Check the file before you process	14
7	Errors	15
	The ones you will actually hit	15
	Full list	15
8	Limits	16
9	Processing notes	17
10	Support	17

1 Start here

You flew somewhere. You know **where** and **when**. Send us that, and get the base-station RINEX for it. You do not need to know station names or browse a coverage map.

Four calls, end to end. Replace the credentials and this runs as-is:

```
BASE=https://ppk.rtkdata.com/api/ppk/v1

# 1. sign in (the token is valid for 7 days)
TOKEN=$(curl -s -X POST $BASE/user/signin \
  -H 'Content-Type: application/json' \
  -d '{"username":"YOUR_USERNAME","password":"YOUR_PASSWORD"}' \
  | jq -r .data.token)

# 2. order: the coordinate you flew at, and the hours you flew (UTC)
ORDER=$(curl -s -X POST $BASE/user/download \
  -H "token: $TOKEN" -H 'Content-Type: application/json' \
  -d '{"longitude":13.405,"latitude":52.52,
      "startYear":2026,"startMonth":7,"startDay":14,"startHour":8,
      "endYear":2026,"endMonth":7,"endDay":14,"endHour":9,
      "fileType":1}')

ORDER_ID=$(echo "$ORDER" | jq -r .data.orderId)
STATION=$(echo "$ORDER" | jq -r .data.stations[0])
echo "$ORDER" | jq .data.baselines    # how far away the base is. Read this.

# 3. poll until done. Stop on success, on failure, and on a deadline:
#    a loop that only waits for success runs forever when an order fails.
DEADLINE=$(( $(date +%s) + 600 ))
while ;; do
  S=$(curl -s $BASE/download/status/$ORDER_ID -H "token: $TOKEN")
  case "$(echo "$S" | jq -r .code)" in
    200) ;;                                # keep going
    *)   echo "status call failed: $S" >&2; exit 1 ;;
  esac
  case "$(echo "$S" | jq -r .data.status)" in
    2) break ;;                            # ready
    3) echo "order failed: $(echo "$S" | jq -r .data.error)" >&2; exit 1 ;;
  esac
  [ "$(date +%s)" -ge "$DEADLINE" ] && { echo "timed out waiting for $ORDER_ID" >&2; exit 1; }
  sleep 5
done

# 4. download the archive, and fetch the base coordinate that your
#    processing software needs
curl -s $BASE/download/$ORDER_ID -H "token: $TOKEN" -o base.zip
file base.zip    # must be a Zip archive; JSON here means read the body, not the file
curl -s $BASE/user/station/$STATION -H "token: $TOKEN" | jq .data.highPrecision
```

Do not write a loop that only waits for success. Poll for a terminal state (2 or 3), read `data.error` when it is 3, and give the loop a deadline. An order that fails will otherwise hold the worker forever and hide the reason.

`base.zip` holds a RINEX observation file and, where the station relays navigation data, a navigation file. Not every station does: if your ZIP has no `.nav`, section 6 explains why and what to use instead. Feed the files, plus the base coordinate from step 4, to Trimble Business Center, RTKLIB, Emlid Studio, Agisoft Metashape, ROCK Desktop, or anything else that reads RINEX.

Billing. Post-processing downloads are included in your RTKdata agreement. There is no per-download charge and no credit balance. Fair-use limits are in section 8.

2 Base URL and conventions

```
https://ppk.rtkdata.com/api/ppk/v1
```

All paths below are relative to that base.

Moved on 2026-10-01. The API used to live at <https://reseller.rtkdata.com/api/ppk/v1>. That address no longer serves it: change the base URL in your configuration. Nothing else changes. Your username and password, any token you hold, your `orderIds` and the station codes all stay valid.

Every response is HTTP 200, including errors

The real outcome is the `code` field in the body, not the HTTP status. A wrong password is HTTP 200 with `{"code": 203}`. Your client must switch on `code`. This is deliberate: it keeps the API a drop-in for the PPK service contract you may already have integrated.

The one exception is the file download, which returns the ZIP body directly on success. Discriminate on `Content-Type: application/zip` means you have your file, `application/json` means read `code`.

Every JSON response uses the same envelope:

```
{ "code": 200, "message": "Success", "data": { } }
```

`data` is omitted on error. **`code` is safe to switch on. `message` is not:** treat it as human-readable text that may change, and never match on it.

Times are UTC, and both hour bounds are inclusive

`startHour: 8, endHour: 9` gives you **two** hours: 08:00:00 through 09:59:59 UTC.

Two things follow, and both cause real failures:

- **Convert your flight times to UTC.** Ordering in local time is the most common way to receive a file that does not overlap your flight.
- **Order the hour before and the hour after your flight.** Base data must fully cover the flight, and hour granularity is coarse. A flight from 09:50 to 10:10 needs `startHour: 9`, `endHour: 10`.

Migrating from another PPK service

Same endpoint paths, same envelope, same order flow, same status codes. Changing the base URL and your credentials is usually the whole migration. Station codes are RTKdata codes, so either re-fetch the station list or use the coordinate flow, which needs no station code at all.

3 Authentication

POST /user/signin

FIELD	TYPE	REQUIRED	DESCRIPTION
<code>username</code>	string	yes	Issued by RTKdata
<code>password</code>	string	yes	

```
{ "username": "YOUR_USERNAME", "password": "YOUR_PASSWORD" }
```

Response [data](#):

FIELD	TYPE	DESCRIPTION
<code>token</code>	string	Valid 7 days

Send it on every subsequent call as a `token` header. Not `Authorization: Bearer:`

```
token: eyJhbGciOi...
```

Codes: [200](#) · [201](#) username or password missing · [202](#) no such user, **or the account is disabled or expired (read message)** · [203](#) wrong password · [305](#) sign-in temporarily unavailable.

When a token expires you get [205](#); sign in again. If you get [202](#) mid-session, signing in again will not help: the account itself is disabled or expired, and [message](#) says which.

GET /user/info

Your account and today's usage. No parameters.

```
{
  "code": 200,
  "message": "Success",
  "data": {
    "username": "dronecustomer",
    "email": "pilot@example.com",
    "createTime": 1784135145871,
    "balance": 0,
    "expense": 0,
    "plan": { "ordersToday": 3, "maxOrdersPerDay": 200, "expiresAt": null }
  }
}
```

FIELD	TYPE	DESCRIPTION
createTime	integer	Account creation, milliseconds since epoch
balance , expense	integer	Always 0 . Present for wire compatibility; RTKdata PPK is not metered per download
plan.ordersToday	integer	Orders placed since 00:00 UTC. Failed orders count
plan.maxOrdersPerDay	integer	Your cap. 201 on the order endpoint means you reached it
plan.expiresAt	integer null	Account expiry in ms since epoch, null if it does not expire

Codes: [200](#) · [201](#) no token · [202](#) account disabled or expired · [205](#) token expired.

4 Ordering data

POST /user/download

Accepts the order and returns an [orderId](#) to poll. The work happens in the background.

Say **either** where you flew, **or** which stations you want:

FIELD	TYPE	REQUIRED	DEFAULT	NOTES
longitude	number	either this pair...		-180 to 180
latitude	number			-90 to 90
count	integer	no	1	Nearby stations to fetch, clamped to 1-5. Multiplies station-hours (section 8)
stations	string[]	...or this		1-5 RTKdata station codes
startYear	integer	yes		UTC
startMonth	integer	yes		1-12
startDay	integer	yes		1-31
startHour	integer	yes		0-23, inclusive
endYear	integer	yes		
endMonth	integer	yes		
endDay	integer	yes		
endHour	integer	yes		0-23, inclusive
fileType	integer	no	1	0 raw RTCM 3, 1 RINEX 3.04, 2 RINEX 2.11
type	integer	no	1	1 all epochs, 2 30-second decimation

If you send both [stations](#) and a coordinate, **[stations wins](#)** and no baseline is reported.

```
{
  "longitude": 13.405,
  "latitude": 52.52,
  "startYear": 2026, "startMonth": 7, "startDay": 14, "startHour": 8,
  "endYear": 2026, "endMonth": 7, "endDay": 14, "endHour": 9,
  "fileType": 1
}
```

Response [data](#):

FIELD	TYPE	DESCRIPTION
orderId	string	24 hex characters. Poll with this
stations	string[]	The station codes we resolved
baselines	array	Only when you ordered by coordinate. Absent if you named stations
baselines[].station	string	
baselines[].distanceKm	number	Distance from the coordinate you sent

```
{
  "code": 200,
  "message": "Success",
  "data": {
    "orderId": "56ec23dbd12cb6d616dbd7df",
    "stations": ["RTKDE465B867C221"],
    "baselines": [ { "station": "RTKDE465B867C221", "distanceKm": 2.01 } ]
  }
}
```

Codes: [200](#) · [201](#) bad parameters, or daily cap reached · [202/205](#) auth · [306](#) window or station-hours too large · [307](#) unknown station code, or no station with data · [305](#) we could not accept the order.

How we choose the station, and what we check

When you send a coordinate, we ask for the nearest stations holding data **at the first hour and at the last hour of your window**, and take the ones present in both.

Know the limit of that check: **a station that drops out and returns inside your window still passes**. We verify the ends, not every hour. If your file looks short, check its first and last epoch (section 6).

If nothing has data near your coordinate you get [307](#). We never silently substitute a distant station.

When quality control withholds a station. Occasionally a station passes the availability check and its data is then withheld by quality control for your window. That only shows when the data is fetched, after your order was accepted. On a coordinate order we chose the station, so we choose again: the order continues with the next nearest station instead of failing. Nothing about that is silent: **stations and baselines in the status response always describe what is in the ZIP**, so read them there once the order is complete. They can differ from what the order response announced, and [distanceKm](#) is the distance to the station you actually received. An order by station code is never changed: if that station is withheld, the order fails with an [error](#) that names it, and [/user/station](#) lists the alternatives nearby.

Read distanceKm before you process

You gave us a coordinate, so you did not choose the base. Baseline length is the strongest single influence on your result: the atmosphere over your rover and over the base stops being the same as they get further apart, and the correction degrades with it.

We deliberately publish no accuracy-versus-distance table, and it is worth being precise about why. Distance-correlated error is real, well understood, and geodetic agencies do model it. What does not exist is a single number or formula that predicts *your* result: the ones in circulation ("1 cm + 1 ppm", "good to 40 km") are vendor conventions for a particular product, not general guarantees, and they do not transfer to your receiver, your antenna, your sky view, your session length and your processing software.

Baseline length is one component of your error budget, not the whole of it. So we give you the component we actually know, the distance, and leave the verdict to the one measurement that is about your flight: **the quality your processing software reports for your solution**. Validate that, and trust it over any table, including one we could have printed here.

If the nearest base is far, order `count: 2` and process both.

5 Stations

Station codes look like `RTKDE465B867C221`. They are stable: a given station always keeps the same code.

GET /user/station

Nearest stations to a point, nearest first.

QUERY PARAM	TYPE	REQUIRED	NOTES
<code>longitude</code>	number	yes	-180 to 180
<code>latitude</code>	number	yes	-90 to 90
<code>year,</code> <code>month,</code> <code>day,</code> <code>hour</code>	integer	no, but pass them	Return only stations holding data for that hour

Always pass the hour. Without it you get the nearest stations in general; with it you get the ones that actually have data. That difference is an order that yields a file versus one that fails.

```
{
  "code": 200,
  "data": {
    "stations": [
      { "name": "RTKDE465B867C221", "distance": 2.01 },
      { "name": "RTKD9CB8B4F0EBF6", "distance": 2.6 }
    ]
  }
}
```

`distance` is in kilometres. An empty `stations` array with `code: 200` means nothing was found: this endpoint does not return `307`.

Codes: `200` · `201` missing or out-of-range coordinate · `202/205` auth · `305` lookup temporarily unavailable.

GET /user/station/{code}

One station, with its coordinates. **This is where you get the base position your processing software needs.**

```
{
  "code": 200,
  "data": {
    "name": "RTKDE465B867C221",
    "status": "ACTIVE",
    "coordinate": {
      "ITRF2020": [3781815.8014, 901079.0687, 5039522.7941],
      "ITRF2020(2015)": [3781815.9893, 901078.8813, 5039522.6887],
      "WGS84": [3781815.7984, 901079.0662, 5039522.7957],
      "local": [3781816.4547, 901078.5066, 5039522.3964]
    },
    "highPrecision": [
      {
        "name": "ITRF2020(2026.50)", "epoch": 2026.5,
        "x": 3781815.8014, "y": 901079.0687, "z": 5039522.7941
      },
      {
        "name": "ETRF2000(2010.00)", "epoch": 2010,
        "x": 3781816.4547, "y": 901078.5066, "z": 5039522.3964
      }
    ]
  }
}
```

FIELD	TYPE	DESCRIPTION
name	string	The RTKdata station code
status	string	As reported by the station, e.g. ACTIVE . Not a fixed set of values
coordinate	object	ECEF metres, keyed by frame. Which keys are present varies by station ; Local in particular is not always there
highPrecision	array	The same positions with the epoch stated explicitly. Prefer this

Codes: [200](#) · [307](#) unknown station code · [201/202/205](#) auth.

Pick a frame, and pick its epoch with it

Coordinates are ECEF metres, offered in several reference frames at several epochs. **A coordinate without an epoch is not a specification.** ITRF2020 at today's epoch and ETRF2000 at 2010.0 describe the same physical antenna and differ by tens of centimetres in Europe, because the ground itself has moved in between.

Mixing frames is the most expensive silent error in this workflow. It raises no error and produces a deliverable that is wrong by an amount too small to notice and too large to accept.

So: **use the frame your project is in**, taken from [highPrecision](#), where the epoch is explicit. If your deliverable is in a national system (ETRS89 in Europe, for instance), use the matching regional entry rather than ITRF at the current epoch.

GET /user/station/list

Every station we carry, with coordinates, in the same per-station shape as above.

This is large: roughly 19,000 stations. **Prefer the nearest-station endpoint.** Use this only if you genuinely need the whole catalogue, and cache it. There is no [distance](#) field here: there is no point to measure from.

6 Polling and the archive

GET /download/status/{orderId}

FIELD	TYPE	DESCRIPTION
status	integer	1 processing · 2 completed · 3 failed. Safe to switch on
orderId	string	

FIELD	TYPE	DESCRIPTION
<code>stations</code>	<code>string[]</code>	
<code>baselines</code>	<code>array</code>	Only for coordinate-resolved orders
<code>fileType,</code> <code>type</code>	<code>integer</code>	As ordered
<code>startTime</code>	<code>string</code>	ISO 8601 UTC. The start of the first hour you ordered
<code>endTime</code>	<code>string</code>	ISO 8601 UTC. The start of the LAST hour you ordered, not the end of coverage. See below
<code>sizeBytes</code>	<code>integer</code>	Present once completed
<code>expiresAt</code>	<code>string</code>	ISO 8601. When the result is deleted
<code>error</code>	<code>string</code>	Present when <code>status</code> is 3. This is your diagnosis. Log it

```
{
  "code": 200,
  "data": {
    "status": 2,
    "orderId": "56ec23dbd12cb6d616dbd7df",
    "stations": ["RTKDE465B867C221"],
    "baselines": [ { "station": "RTKDE465B867C221", "distanceKm": 2.01 } ],
    "fileType": 1,
    "type": 1,
    "startTime": "2026-07-14T08:00:00.000Z",
    "endTime": "2026-07-14T09:00:00.000Z",
    "sizeBytes": 4699566,
    "expiresAt": "2026-07-16T16:53:52.000Z"
  }
}
```

`endTime` echoes what you ordered; it does not describe the archive. For the order above, `endHour`: 9 echoes back as `endTime`: 09:00:00Z, while the data actually runs to 09:59:59Z, because the end hour is inclusive. Both fields are the hour *labels* you sent, not coverage boundaries.

So do not compute coverage from `startTime/endTime`. **The authority on what you received is the RINEX file itself:** its `TIME OF FIRST OBS` and `TIME OF LAST OBS` headers. Check those against your flight (section 6). We are aware this is a poor field name; it is kept for wire compatibility, and a future version will use an unambiguous half-open interval.

One station-hour typically completes in **15 to 30 seconds**. Larger orders take proportionally longer, and orders are processed one at a time, so a queue adds to that. Poll every 5 seconds, and give your loop a deadline.

An unknown `orderId`, or one belonging to another account, returns 201.

Codes: 200 · 201 unknown orderId, or no token · 202/205 auth. This endpoint never returns 304.

GET /download/{orderId}

Returns the ZIP on success; JSON on anything else. Discriminate on `Content-Type`.

Codes: 200 ZIP body · 304 not finished, keep polling · 305 the order failed (`message` says why), **or the result expired and was deleted** · 201 unknown orderId · 202/205 auth.

What is in the archive

Files are named `{station}_{YYYYMMDD-HH}.{ext}`, where the timestamp is the **start** of the window.

ORDER	FILES
<code>fileType: 1</code> (RINEX 3.04)	RTKDE465B867C221_20260714-08.obs, RTKDE465B867C221_20260714-08.nav
<code>fileType: 2</code> (RINEX 2.11)	the same two, with RINEX 2.11 content
<code>fileType: 0</code> (raw RTCM)	RTKDE465B867C221_20260714-08.rtc3: one file, no navigation file
<code>count: 2</code> , or several <code>stations</code>	one .obs per station, plus navigation (see below)

You may receive fewer navigation files than stations. That is correct, not a fault.

A navigation file holds **broadcast ephemeris**: where the satellites were. It describes the satellites, not the base station, so the same message received at two different stations carries the same content. That is why one navigation file can serve every station in your order, and why some stations produce none at all: they do not relay the ephemeris messages, so you use the file from another station.

One qualifier, and it matters. "Station-independent" is about the *content*, not the *completeness*: each receiver only records the satellites it actually tracked, so a navigation file is not guaranteed to cover every constellation in every observation file. Before processing, check that the navigation file carries the constellations your .obs uses (section 6). If it is short, take one from another station in the order, or from a public broadcast-ephemeris archive; a merged daily file from any GNSS data centre works and is not specific to our network.

Check the file before you process

Thirty seconds here saves an afternoon. Open the `.obs` in a text editor:

```

3.04          OBSERVATION DATA      M: Mixed          RINEX VERSION / TYPE
RTKDE465B867C221          MARKER NAME
3781815.4669  901079.1274  5039522.6662          APPROX POSITION XYZ
G  12 C1C L1C S1C C2W L2W S2W C2X L2X S2X C5Q L5Q S5Q          SYS / # / OBS TYPES
R   6 C1C L1C S1C C2C L2C S2C          SYS / # / OBS TYPES
2026    07    14    08    00    00.00000000          GPS          TIME OF FIRST OBS
2026    07    14    08    59    59.00000000          GPS          TIME OF LAST OBS

```

Confirm three things:

1. **TIME OF FIRST OBS and TIME OF LAST OBS fully cover your flight.** Not overlapping is the single most common reason post-processing fails. This check is about more than the inclusive-end-hour nuance above: it is your defence against a file dated to the wrong day entirely, from any provider's archive or converter fault. We also enforce it server-side: before an order completes, we verify the delivered dates against the window you ordered, and fail the order with an explicit error rather than deliver a mismatched file. Keep your own check anyway; it costs seconds and it is the one that runs on what actually landed on your disk.
2. **The `SYS / # / OBS TYPES` lines list the constellations you expect** (G GPS, R GLONASS, E Galileo, C BeiDou).
3. **The epoch interval is what you ordered.** `type: 1` gives the station's native rate, typically 1 Hz. `type: 2` gives one epoch per 30 seconds.

Use `type: 2` only if your software specifically wants a low-rate base. **For kinematic drone work keep `type: 1`:** several drone post-processing tools reject or degrade a 30-second base, and the saving in file size is not worth the risk.

Do not use `APPROX POSITION XYZ` as your base coordinate. It is not only less precise: the RINEX header states no reference frame and no epoch for it, so you cannot know what it means. A coordinate without a frame and an epoch is not a specification (section 5). Take the base position from `/user/station/{code}`, where both are explicit.

7 Errors

The ones you will actually hit

CODE	MEANING	DO THIS
205	Token expired	Sign in again
307	No station with data for that place and window	Widen the hours, check you converted to UTC, or the window predates the archive (section 8)
306	Order too large	Split it (section 8)
305 on download	The order failed, or the result expired	Read message . If it expired, re-order
201 on download or status	Unknown orderId	Check the id you stored

Full list

CODE	MEANING
200	Success
201	Parameter error, token missing, unknown orderId, or daily cap reached
202	No such user, or the account is disabled or expired (read message)
203	Wrong password
205	Token expired or malformed
304	Order not finished (download endpoint only)
305	Order failed, result expired, or a service fault
306	Window or station-hours too large
307	Unknown station code, or no station with data

[code](#) is the contract. [message](#) is prose and may change.

8 Limits

LIMIT	VALUE
Station-hours per order	12
Stations per order	5
Orders per day	200 by default, per account. Check yours at /user/info
Result retention	Guaranteed until you download it once , then best-effort up to 24 hours. See below
Archive depth	At least 12 months , bounded per station by when it came online. See below

Station-hours is the limit that binds. It is `stations x hours`, with both hour bounds inclusive:

ORDER	STATION-HOURS	
1 station, 14 Jul 08:00-19:00	1 x 12 = 12	accepted
1 station, 14 Jul 08:00-20:00	1 x 13 = 13	306
2 stations, 6 hours	2 x 6 = 12	accepted
<code>count: 2</code> , 8 hours	2 x 8 = 16	306

A flight is minutes to hours, so this is rarely in your way. If your workflow needs more, ask: these are policy limits, not technical ones.

Failed orders count against your daily cap. Retrying a bad window repeatedly costs quota.

Retention, stated honestly. There are two different promises here, and only one of them is a promise:

- **Until your first successful download, the result is kept.** A result you have not fetched is never removed early. This is the guarantee you can build on.
- **After that, up to 24 hours, best effort.** Once you have downloaded a result, it becomes a candidate for early removal if storage runs tight, oldest first.

`expiresAt` is therefore the **latest** a result can live, not a reservation. Do not schedule a second retrieval against it and assume the file will be there. Download once, keep your copy. If you need it again, re-order: it costs one slot from your daily allowance and takes the usual 15 to 30 seconds.

Orders are never deduplicated: ordering the same window twice costs two slots and fetches twice.

Archive depth. At least 12 months: any window up to a year old can be ordered on demand, and we verify that by live retrieval, not by assumption. The natural bound is the station itself: recording starts the day a station comes online, so a station younger than a year carries its full history rather than 12 months. Whether a specific station has data for a specific historical hour is exactly what the nearest-station lookup answers when you pass the hour (section 5). If your reprocessing horizon is longer than 12 months, contact us before you depend on it and we will confirm the depth for your case. Whatever the depth, the failure mode is honest: a window the archive cannot serve fails, with `307` at order time when we resolve the station from your coordinate, or `status: 3` with an explanatory `error` otherwise. A completed order always contains data dated to the window you ordered, never a silent substitute.

Two practical consequences for a pipeline design. Fetch base data when you ingest the flight, not when you reprocess it: the flight's window only gets older, and your own storage is the only retention you control (see above; results are best-effort after first download too). And if you plan to reprocess old flights, keep the base ZIP with the flight, which also pins the exact observations your deliverable was built from.

9 Processing notes

Take the base coordinate from `/user/station/{code}`, in the frame your project uses, with its epoch. Not from the RINEX header. This is the single biggest determinant of your absolute accuracy, and taking it from the wrong place, or the wrong frame, is the most common way to produce a deliverable that is wrong by a repeatable, plausible-looking offset.

Antenna height is the classic blunder. Whatever you enter for your rover, be clear whether it is measured to the antenna reference point or to the phase centre, and whether your software adds the offset for you. Do not count it twice. Nothing about the base's antenna height needs entering on your side: it is already accounted for in the base data.

Leave the delivered RINEX headers as they are. Your software reads them.

Baseline length is covered in section 4. Order `count: 2` and process both if the nearest base is far.

Time is UTC throughout the API. RINEX observation epochs are GPS time, as usual.

10 Support

`jonas@rtkdata.com`. Include the `orderId` and the **station code**: those two are enough for us to trace your exact request, including what we asked upstream and what came back.

If an order failed, send the `error` string from `/download/status/{orderId}` with it.