BST-HSDAQ6 Communication Protocol 1.1

Boston Smart Technologies · BST-HSDAQ6 data logger · November 17, 2026

Overview and quick start

The BST-HSDAQ6 logger is controlled with short text commands, sent the same way over Bluetooth Low Energy or a wired USB serial port. Every command and reply in this document is byte-for-byte identical on both links; only the way you connect differs (next section).

The logger records up to 6 differential analog channels to its microSD card. A client can configure it, start or arm a recording, download the files, and watch a slow live view of the inputs. The logger describes itself (identity with `INFO`, capabilities in the `limits` block of `GETCFG`), so a client should never hard-code channel counts, rates or limits.

Quick start with the reference client `bst_hsdaq6.py`, in the `python` folder of the developer kit (Python 3, `pip install -r requirements.txt`):

1. `python bst_hsdaq6.py --ble info`: the logger answers with its protocol version, part number and serial number. Use `--port COM5` (or `/dev/ttyACM0`) instead of `--ble` for the USB port.
2. `python bst_hsdaq6.py --ble time --set`: sets the logger clock to your computer's.
3. `python bst_hsdaq6.py --ble capture`: records one file with the current settings.
4. `python bst_hsdaq6.py --ble list`, then `python bst_hsdaq6.py --ble get DATA0001.BIN`: downloads the file and writes a CSV in engineering units.

To talk to the logger from a terminal program over USB, open the port and type `INFO` followed by Enter.

Connecting

Bluetooth Low Energy

The logger uses the Nordic UART Service, a standard BLE serial pipe supported by most BLE libraries and phone apps (nRF Connect, Serial Bluetooth Terminal).

Item	Value
Advertised name	starts with `BST-BLE` (`BST-BLE_<device_id>` when a device id is set)
Service UUID	`6E400001-B5A3-F393-E0A9-E50E24DCCA9E`
Write commands to (RX)	`6E400002-B5A3-F393-E0A9-E50E24DCCA9E`, write without response
Replies arrive on (TX)	`6E400003-B5A3-F393-E0A9-E50E24DCCA9E`, notify: enable notifications first
Largest write	180 bytes, or MTU − 3 if the MTU is smaller; split longer commands across writes
Largest notification	217 bytes
Pairing	none

Treat the notifications as one continuous byte stream: a reply line or a data block can be split across notifications, and one notification can hold the end of one reply and the start of the next. A long command (such as `PUTCFG`) may be sent as several writes back to back, up to about 2 kB in total.

Only one Bluetooth client can be connected at a time. While connected, the logger stops advertising.

USB serial port

The logger appears on the computer as a USB virtual COM port (CDC ACM): COMn on Windows, `/dev/ttyACMn` on Linux, `/dev/cu.usbmodem…` on macOS. No driver is needed on current Windows, Linux or macOS.

- Baud rate, parity and stop bits are ignored by the USB port; any setting works.
- Open the port with DTR asserted (the default for most libraries and terminals). Closing the port, or dropping DTR, ends any live stream.
- Throughput is much higher than Bluetooth, so this is the preferred link for downloading large recordings.

The USB port is on hardware revision C and later. Earlier units have Bluetooth only; `INFO` reports the revision as `hw_rev`.

Using both links at once

Both links accept commands at the same time. Commands run one at a time in the order they arrive, and each reply goes back on the link that sent the command. A live stream goes to the link that started it. Two clients changing settings at once will confuse each other, so connect one controlling client at a time.

Message rules

A command is one line of ASCII text ending in LF (`\n`); a CR before the LF is ignored. Commands and keywords are upper case and case sensitive. Arguments are separated by single spaces. The longest accepted line is 4,599 bytes; a longer line is answered with `ERR`.

Every reply line ends in CR LF (`\r\n`). Each command ends with exactly one terminator line, so a client always knows when the reply is complete:

Reply shape	Commands	Ends with
Status only	`ARCH`, `DEL`, `ARM`, `STOP`, `REARM`, `CAPTURE`, `STREAM`, `SETCAL`, `TIME <set>`, `AUTH <answer>`, `SETKEY`, `ZERO CLEAR`	`OK` or `ERR`
Data lines, then status	`INFO`, `STATUS`, `TIME`, `ERASE`, `AUTH`, `ZERO`	`OK` (or `ERR` alone)
List	`LIST`	`END` (or `ERR` alone)
Configuration JSON	`GETCFG`, `PUTCFG`, `TUNE`	JSON text, then a line `END` (or `ERR` alone)
Binary block	`GET`	`SIZE <n>`, then exactly n raw bytes

Send one command and wait for its terminator before sending the next. Bytes sent while a reply is still going out may be lost.

Status lines (starting with a hash sign)

A line starting with `#` is a status message for people (for example `# Saved DATA0007.BIN` or `# Download: DATA0007.BIN 40%`). It is never part of a reply and never arrives inside one. Clients skip these lines. They appear on the USB serial port only, never over Bluetooth.

Errors

Protocol 1.x has one error reply: a line that is exactly `ERR`. It means one of: unknown command or bad arguments; the logger is locked by its owner key (see Access control; `INFO` shows `session=locked`); the logger is busy (recording, or between files of a continuous run, see `STATUS`); no SD card or a card error; the file does not exist. `GET` signals failure with `SIZE 0` instead. The error line stays a bare `ERR` in every 1.x version, so a client can match it exactly.

Busy rules

Logger state (`STATUS`)	Allowed	Refused with `ERR`
`idle`	everything	—
`armed`, `recording`	`STATUS`, `STOP`, `INFO`, `TIME`, `GETCFG`	file commands, settings, `CAPTURE`, `ARM`, `STREAM ON`, `ZERO`
`hold` (continuous run, between files)	file commands, `STATUS`, `STOP`, `REARM`, `INFO`, `TIME`, `GETCFG`	`PUTCFG`, `TUNE`, `CAPTURE`, `ARM`, `STREAM ON`, `ZERO`

On connecting

A client should send `STREAM OFF`, wait about 0.3 s and discard everything received, in case an earlier session left a live stream running. `STREAM OFF` is harmless at any time. Then send `INFO`: if it shows `session=locked`, unlock with `AUTH` (next section) before anything else.

Access control

A logger's owner can choose to set an **owner key**, a simple safeguard against someone in Bluetooth range controlling it. With no key set (as shipped) every command works and this section can be ignored. With a key set, each new connection, Bluetooth or USB, starts **locked**: only `INFO`, `STATUS` and `AUTH` work, and every other command answers `ERR` until the session proves it knows the key. The key itself never travels: the logger sends a random challenge and the client answers with a value only the key can produce.

The key. The owner chooses a passphrase (8 characters or more). The owner key is the SHA-256 hash of the passphrase in UTF-8: 32 bytes.

Unlocking:

1. Send `AUTH`. The logger answers `nonce=<32 hex digits>` (16 random bytes), then `OK`.
2. Compute HMAC-SHA256 with the owner key as the key and the 16 nonce bytes (not the hex text) as the message.
3. Send `AUTH <the 32-byte result as 64 hex digits>`. The logger answers `OK`: this connection is unlocked until it ends. Or `ERR`: wrong answer.

Each nonce is good for one answer; request a new one for every try. After 3 wrong answers in a row, the logger ignores `AUTH` for 1 s, then 2, 4 ... up to 60 s after each further wrong answer. A right answer resets this. Disconnecting locks the logger again.

Setting, changing and removing the key:

Command	Where	Effect
`SETKEY <64 hex digits>`	any link while no key is set; afterwards an unlocked session	Sets the owner key (the SHA-256 of the new passphrase), or replaces it. A new owner sets it over Bluetooth right away.
`SETKEY`	an unlocked session	Removes the key: the logger is open again.

Both answer `OK` or `ERR`, and `ERR` while recording or armed. The connection that set or changed the key stays unlocked. The passphrase itself never travels; its SHA-256 does, once, when the key is set.

Where the key is kept. In a file `OWNER.KEY` on the logger's SD card, scrambled so it isn't readable as text and doesn't work in another logger. It never appears in `LIST` and `ERASE` leaves it alone. **To reset a logger whose passphrase is forgotten, delete `OWNER.KEY` or wipe the card** in a computer: the logger is open again from the next connection. A logger with no card has no key.

What it protects. A locked logger can't be controlled, read, configured or wiped over Bluetooth by someone who doesn't have the key. It is a safeguard, not high security: it is not encryption (someone listening to an unlocked session, or to the moment the key is set, can still see what goes past), and anyone who can take the SD card out can remove the key.

```
> INFO
protocol=1.1
...
auth=required
session=locked
OK
> AUTH
nonce=d3a53cbd167130347d75193bb8b8beda
OK
> AUTH 5f0c... (64 hex digits)
OK
```

In Python, with `key = hashlib.sha256(passphrase.encode()).digest()`, the answer is `hmac.new(key, bytes.fromhex(nonce), hashlib.sha256).hexdigest()`.

Command reference

Identity and clock

Command	Reply	Notes
`INFO`	`key=value` lines, then `OK`	Keys in 1.0: `protocol` (major.minor), `part_number` (`BST-HSDAQ6`), `hw_rev`, `fw`, `serial_number`, `mcu_uid` (24 hex digits, unique per unit), `ble_mac`, `device_id`, `channels`, `auth` (`none` or `required`: whether an owner key is set), `session` (`locked` or `unlocked`: this connection), `link` (`ble` or `serial`: the link this reply came on). Ignore keys you do not know; later versions may add some. Firmware before 1.0 answers `ERR`: treat it as protocol 1.0 without identity.
`TIME`	`YYYY-MM-DD HH:MM:SS`, then `OK`; or `NOT SET`, then `OK`	The logger clock keeps running on its coin cell. It has no time zone: it holds whatever local time you set.
`TIME YYYY-MM-DD HH:MM:SS`	`OK`	Sets the clock. Recordings are time-stamped from it.

`AUTH`	`nonce=<32 hex>`, then `OK`	A new challenge (see Access control). With no owner key set: `OK` alone.
`AUTH <64 hex>`	`OK` or `ERR`	Answer to the last challenge; `OK` unlocks this connection.
`SETKEY <64 hex>`	`OK` or `ERR`	Set the owner key (allowed while none is set), or change it (unlocked session).
`SETKEY`	`OK` or `ERR`	Remove the owner key. Needs an unlocked session.

Settings

Command	Reply	Notes
`GETCFG`	configuration JSON, then `END`	See Configuration.
`PUTCFG <json>`	the configuration as stored, then `END`	Replaces the whole configuration: keys you leave out return to their defaults. The logger clamps values to its limits; read the echo to see what it kept (for example the achieved sample rate). Saved to the SD card.
`TUNE`	as `GETCFG`	Re-plans the sample rate from the stored settings.

Recording

Command	Reply	Notes
`CAPTURE`	`OK` when the file is closed, `ERR` if it failed	Records one file now, whatever the capture mode. The reply can take as long as the recording.
`ARM`	`OK` at once	Waits for the trigger set in the configuration. With `trigger.rearm` = `single`: one file, then idle. With `continuous`: after each file the logger enters `hold`, then re-arms. Poll `STATUS`.
`STATUS`	`<state> <rearm> <files>`, then `OK`	state: `idle`, `armed`, `recording` or `hold`. rearm: `single` or `continuous`. files: files saved since the last `ARM`.
`REARM`	`OK`, or `ERR` if not in `hold`	Ends the hold and arms for the next file. Without it the logger re-arms by itself after `ble_window_s` seconds with no download (at once if no client is connected).
`STOP`	`OK`	Disarms, or ends a recording early (the file keeps what was recorded), and ends a continuous run.

A continuous run, in order: `ARM`; poll `STATUS` until the file count rises and the state is `hold`; `LIST` and `GET` the new file; `REARM`; repeat. `STOP` ends it.

Files

Recordings are named `DATAnnnn.BIN` (nnnn = 0001–9999) and kept in the card's root. Archived files are in `SENT/`; write their names as `SENT/DATA0007.BIN`.

Command	Reply	Notes
`LIST NEW`	`<name>,<size>,<seq>` per file, then `END`	Files in the root. size in bytes, seq = nnnn.
`LIST SENT`	as above	Archived files.
`GET <name>`	`SIZE <n>`, then n raw bytes	The whole file. `SIZE 0` = not found or busy.
`GET <name> <offset> <length>`	`SIZE <n>`, then n raw bytes	A byte range: use it to download in pieces and to resume after a dropped link. length 0 = to the end; a range past the end is shortened.
`ARCH <name>`	`OK`	Moves a file to `SENT/`.
`DEL <name>`	`OK`	Deletes a file (`SENT/` names allowed).
`ERASE`	`<count>`, then `OK`	Deletes every recording, root and `SENT/`. The configuration file stays. On `ERR` it may be part done: `LIST` shows what is left.
`SETCAL <name> <json>`	`OK`	Rewrites the channel names, units and calibration stored in an existing file. The JSON has the same `channels` array as the configuration; other keys are ignored. The samples are untouched.

Live stream

Command	Reply	Notes
`STREAM ON <rate>`	`OK`, then binary packets	rate in samples per second per channel; 0 or absent = the default; capped at `limits.stream_max_hz`. Only while idle.
`STREAM OFF`	`OK`	Packets already queued arrive before the `OK`.

Starting a recording, or `ZERO`, ends a live stream.

Zero (protocol 1.1)

Zeroing measures each channel's offset with its inputs at their zero condition (shorted, or the sensor at rest) and keeps it on the logger. The logger takes it with exactly the sampling a recording uses at the configured sample rate, so the offset it finds is the one your recordings have.

Command	Reply	Notes
`ZERO`	`ZERO <rate_hz> <v1> … <vC>`, then `OK`; or `ERR`	Measures 5 runs of 0.5 s of data, each after the usual start-up discard, and saves their average. rate_hz is the achieved sample rate it was taken at; v1 to vC are each channel's offset in volts at the input. Only while idle. The reply comes when all 5 runs are done: each run is the 1,000-sample start-up discard plus 0.5 s, so the whole zero takes about 4 s at 5 kHz and 8 s at 1 kHz.
`ZERO <samples>`	as `ZERO`	The same, with that many samples per channel in each run (at most 1,000,000).
`ZERO CLEAR`	`OK`	Removes the zero.

- A new `ZERO` replaces the old zero; it never builds on it. Repeating it with the same inputs gives the same values, within noise.
- The zero is subtracted from the samples of each new file as they are recorded, and from the live stream. Each file's calibration entry keeps your own `offset` unchanged. Because the zero is in the samples, `SETCAL` cannot remove it from a file: to correct a wrong zero, zero again and record again.
- Files recorded before a `ZERO`, or after `ZERO CLEAR`, are unchanged.
- `ZERO` also refreshes the logger's ADC calibration and its mid-scale reference, and later recordings reuse both, so they sample exactly as the zero did. Both are redone at power-up, after which the kept zero matches less closely (by up to a few mV). For best accuracy, zero again after power-up once the logger has warmed up for a few minutes, and after large temperature changes.
- The offset can depend on the sample rate. If you change `target_sample_rate_hz`, zero again: compare `zero.rate_hz` with `daq.achieved_rate_hz` to see whether the zero was taken at the current rate.
- `ERR` if the logger is busy, or if one run would take longer than 10 s (sample rates below about 100 Hz, where the start-up discard alone is long).
- The live view samples the inputs differently from a recording, so a small part of the offset can remain in it after zeroing. Recordings don't have this: they sample exactly as the zero did.

Configuration

The configuration is one JSON object, read with `GETCFG` and written with `PUTCFG`. You own the settings below; the logger owns the `daq`, `limits`, `armed` and `zero` keys and ignores them in a `PUTCFG`. Unknown keys are ignored. Always read `GETCFG` first, change what you need, and send the whole object back.

Key	Type	Meaning
`target_sample_rate_hz`	number	Samples per second, per channel; all channels are sampled together.
`total_samples`	integer	Samples per channel in each file. With a trigger: pre-trigger plus post-trigger.
`pre_samples`	integer	Samples kept from before the trigger. Clamped to `limits.max_pre_samples`.
`capture_mode`	string	What the logger's button does: `app` (records at once), `trigger_roll` (arms the trigger), `boot` (records at power-up). `CAPTURE` and `ARM` work in every mode.
`trigger.channel`	integer	Trigger channel, 0-based (0 = channel 1).
`trigger.slope`	string	`rising` or `falling`.
`trigger.threshold_volts`	number	Level in volts at the input, before calibration. With a zero set, it is compared with the zeroed input, the same values the file holds.
`trigger.rearm`	string	`single` or `continuous` (see `ARM`).

`ble_window_s`	integer	Continuous runs: seconds the logger holds for a download before re-arming.
`device_id`	string	Short label, also shown in the Bluetooth name. Up to `limits.device_id_max` characters.
`channels[]`	array	Per channel: `name`, `unit`, `units_per_volt`, `offset`. Engineering value = volts × `units_per_volt` + `offset`, where volts are the input volts with the zero already subtracted. Copied into every new file unchanged.

Defaults on a new SD card: 1000 Hz per channel, 5000 samples per file (5 s), 500 pre-trigger samples, trigger on channel 1 rising through 1.000 V, single re-arm, button records at once.

Logger-owned, read only:

- `daq.achieved_rate_hz` is the sample rate actually used, which can differ slightly from the request; `daq.tune_status` says `exact`, `nearest`, `clamped` or `untuned`.
- `armed` is true while the logger is armed or in a continuous run; it re-arms after a power cut.
- `zero` (1.1) is the zero kept by `ZERO`: `{"rate_hz": 5000.000, "volts": [-0.003296, 0.00419, …]}`, one entry per channel, volts at the input. `rate_hz` 0 means no zero. Only `ZERO` and `ZERO CLEAR` change it; a 1.0 logger has no `zero` key.
- `limits` is what this logger can do. Build your user interface from it: `max_channels`, `channel_bits` (native ADC bits per channel), `min_rate_hz`, `max_rate_hz`, `min_samples`, `max_samples`, `max_pre_samples` (at the current rate), `device_id_max`, `name_max`, `unit_max`, `ble_window_min_s`, `ble_window_max_s`, `stream_max_hz`, `capture_modes`, `sleep_modes`.

Example `PUTCFG` (one line, shortened):

```
PUTCFG {"target_sample_rate_hz":10000,"total_samples":50000,"pre_samples":5000,"capture_mode":"trigger_roll","trigger":
{"channel":0,"slope":"rising","threshold_volts":1.5,"rearm":"single"},"channels":
[{"name":"Pressure","unit":"psi","units_per_volt":100,"offset":0}, ...]}
```

Live stream packets

After `STREAM ON` answers `OK`, the logger sends binary packets about 10 times a second until `STREAM OFF`. All numbers are little-endian.

Offset (bytes)	Size	Field
0	4	Sequence number, unsigned, +1 per packet sent, from 0
4	2	n = sample instants in this packet, unsigned (1–16 in this version)
6	1	c = channels per instant, unsigned (6)
7	1	Always 0
8	n × c × 2	Samples: signed 16-bit, millivolts at the input (before calibration, with the zero subtracted), instant by instant, channel 1 first

Reading them:

- Read 8 bytes, check that byte 7 is 0 and that n and c are in range, then read n × c × 2 bytes. If the check fails, drop one byte and try again: this finds the next packet after any lost bytes.
- A jump in the sequence number means packets were dropped (the link was too slow). The instants inside a packet are evenly spaced at 1 / rate.
- After you send `STREAM OFF`, keep reading packets until the next 4 bytes are `OK\r\n`.
- The live view is for watching signals, not for measurement: use recordings for that. The rate is capped at `limits.stream_max_hz` (2,000 per second in this version).

Recording file format (version 3)

A recording is one binary file: a 64-byte header, a 32-byte calibration entry per channel, then the samples. `GET` returns exactly these bytes, so the same reader works on files copied off the SD card. All numbers are little-endian; `f32` is an IEEE 754 single.

Header

Offset	Size	Type	Field
0	12	ASCII	Signature `BlueNRG-DAQ` followed by one zero byte
12	2	u16	Format version: 3

14	2	u16	C = channels
16	4	u32	N = samples per channel
20	4	f32	Sample rate, Hz per channel (the requested rate)
24	4	u32	Sequence: the nnnn of `DATAnnnn.BIN`
28	2	u16	Bits per sample: 16
30	2	u16	Sample format: 0 = unsigned
32	4	i32	code_min: 0
36	4	i32	code_max: 65535
40	4	f32	v_min: volts at the input for code_min
44	4	f32	v_max: volts at the input for code_max
48	4	u32	Trigger index: sample number of the trigger (triggered files)
52	2	u16	Capture method: 0 = started directly, 1 = triggered
54	2	u16	Source: 0 = ADC
56	4	u32	CRC-32 of the samples only (the same CRC as zlib / Ethernet)
60	4	u32	Start time: the logger's local clock in seconds since 1970-01-01, counted as if it were UTC; 0 = clock not set

Calibration entries (C × 32 bytes, from offset 64)

Offset in entry	Size	Field
0	16	Channel name, ASCII, zero-padded
16	8	Unit, ASCII, zero-padded
24	4	f32 units_per_volt
28	4	f32 offset (the configured offset)

Samples (from offset 64 + 32 × C)

N × C unsigned 16-bit codes, interleaved: sample 0 of channels 1 to C, then sample 1, and so on. With a zero set (protocol 1.1), the codes already have it subtracted. Channels with a narrower ADC are scaled up to 16 bits, so every channel uses the same formulas:

```
volts = v_min + code × (v_max − v_min) / (code_max − code_min)
value = volts × units_per_volt + offset
```

Sample i was taken at i / rate seconds after the start; in a triggered file, subtract the trigger index to put the trigger at 0 s. For exact timing over long files, use `daq.achieved_rate_hz` from `GETCFG`, which can differ slightly from the requested rate in the header.

Check the CRC after a download: a mismatch means the file is damaged or incomplete. Readers should refuse files whose signature or version they do not know.

Worked examples

Lines starting `>` are sent by the client; the rest are the logger's replies.

Identify the logger:

```
> INFO
protocol=1.1
part_number=BST-HSDAQ6
hw_rev=C
fw=1.0.0
serial_number=2647-0001
mcu_uid=34353932303551190026001F
ble_mac=F1:84:24:A6:EE:A3
device_id=DAQ1
channels=6
OK
```

Record one file and download it in two pieces:

```
> CAPTURE
OK
> LIST NEW
DATA0001.BIN,1200256,1
END
> GET DATA0001.BIN 0 65536
SIZE 65536
<65,536 bytes>
> GET DATA0001.BIN 65536 0
SIZE 1134720
<1,134,720 bytes>
> ARCH DATA0001.BIN
OK
```

Zero the inputs (shorted), then record:

```
> ZERO
ZERO 5000.000 -0.003296 0.004190 -0.002315 0.004787 -0.000995 0.010767
OK
> CAPTURE
OK
```

Wait for a trigger:

```
> ARM
OK
> STATUS
armed single 0
OK
> STATUS
idle single 1
OK
```

Reference client

`bst_hsdaq6.py` is a single Python 3 file that implements everything in this document; copy from it freely. It needs `bleak` for Bluetooth and `pyserial` for the USB port, and uses `numpy` when installed. `test_bst_hsdaq6.py`, next to it, checks the client offline against a simulated logger, or read-only against a real one (`--ble` or `--port`).

```
from bst_hsdaq6 import HsDaq, parse_recording, save_csv

with HsDaq.open_ble() as daq:                 # or HsDaq.open_serial("COM5")
    print(daq.check_protocol())
    daq.unlock("my passphrase")               # only if the owner set a key
    daq.set_time()
    daq.zero()                                # inputs shorted or at rest (1.1)
    daq.capture()
    name = max(daq.list_files(), key=lambda f: f["seq"])["name"]
    rec = parse_recording(daq.download(name))
    save_csv(rec, "recording.csv")
```

Versioning

`INFO` reports the protocol version as major.minor. A new minor version only adds: new commands, new `INFO` keys, new configuration or `limits` keys. A client written for 1.0 keeps working with any 1.x logger if it ignores keys it does not know and skips lines starting with `#`. A new major version may change existing commands; check the major number on connecting and stop with a clear message if it is not one you support.

The recording file carries its own format version (3 in this document), which changes independently.

This document and the logger firmware always state the same protocol version.

Version	Date	Changes
1.0	2026-11-15	First published version: `INFO`, Bluetooth and USB serial links, owner key (`AUTH`, `SETKEY`), all commands documented
1.1	2026-11-17	Zeroing: `ZERO` (average of 5 runs; refreshes the ADC calibration that recordings reuse), `ZERO CLEAR`, the logger-owned `zero` configuration key; the zero is subtracted from recorded samples, the live stream and the trigger level