

BlueFly Vario — BLE Integration Guide for App Developers

Audience: Android and iOS developers building apps that connect to a BlueFly Vario device. **Scope:** Everything needed to discover, connect, stream telemetry, read/write settings, and download flight logs over Bluetooth Low Energy. **Derived from:** BlueFly reference Android app (com.blueflyvario.blueflybridge, v1.1.1) and the *BlueFlyVario Hardware Settings Manual* v2.0 (Jan 2024, covering v22/v23 models). Where the two differ, firmware behaviour follows the manual; the app shows one concrete client implementation.

This document describes the *wire protocol* and BLE behaviour, so it applies equally to native Android (Kotlin/Java), native iOS (Swift/CoreBluetooth), Flutter, React Native, or any BLE-capable platform. Code is kept to the minimum needed to remove ambiguity.

Companion document. This guide is intended to be read alongside the *BlueFlyVario Hardware Settings Manual*. The manual is the authoritative reference for what each setting and command *means* and its valid range (Annex A); this guide describes how to reach them *over BLE* from application code. Where this guide gives a scale factor, range, or default, the manual’s Annex A prevails.

Note: This guide is AI generated as of July 2026 and may not be 100% accurate.

1. How the device presents itself

The BlueFly is, at the transport level, a **transparent serial-over-BLE bridge**. It exposes a proprietary UART-style GATT service (the Microchip/ISSC “transparent UART” profile). Once you subscribe to notifications and enable writes, the link behaves like a bidirectional ASCII serial port:

- **Everything the device sends you** is line-oriented ASCII text terminated by CR (\r, 0x0D) and/or LF (\n, 0x0A).
- **Everything you send the device** is an ASCII command string.

There is no binary packet format to decode. All telemetry, settings, and log data are human-readable text. This makes the device very easy to integrate and debug — a raw terminal view of the stream is usually enough to see what is happening.

2. BLE transport layer

2.1 GATT service and characteristics

Role	UUID	Properties	Direction
Service	49535343-FE7D-4AE5-8FA9-9FAFD205E455	—	—

Role	UUID	Properties	Direction
TX characteristic	49535343-1E4D-4BD9-BA61-23C647249616	Notify	Device → App
RX characteristic	49535343-8841-43F4-A8D4-ECBE34729BB3	Write Without Response	App → Device
CCCD descriptor	00002902-0000-1000-8000-00805F9B34FB	—	Enables notifications on TX

Naming is from the device's point of view: the device **transmits** telemetry on the **TX** characteristic (which the app subscribes to) and **receives** your commands on the **RX** characteristic (which the app writes to).

2.2 Discovery / scanning

BlueFly devices advertise a friendly device name (typically beginning with Bluefly). The reference app does **not** filter the scan by service UUID — it performs a general low-latency LE scan and filters candidates by advertised name, discarding any result with a null/blank name.

Recommended approach: - Scan with low-latency settings for responsiveness. - Filter by advertised name prefix, or present all named devices and let the user pick. - Alternatively you *can* filter by the service UUID in section 2.1 if you prefer a stricter scan; both work. - Impose a scan timeout (the reference app uses 30 s) and always stop the scan before connecting.

2.3 Connection sequence

Order matters. The reference implementation follows this exact sequence, and both platforms benefit from it:

1. **Connect** to the device (on Android: connectGatt with TRANSPORT_LE; autoConnect = true is used to survive transient dropouts).
2. **Request a large MTU.** The app requests **512 bytes**. A large MTU materially improves throughput for the 50 Hz raw stream and for bulk IGC downloads.
3. **Request a high connection priority** (short connection interval, ~7.5 ms) immediately after the MTU is negotiated.
4. **Wait briefly** (~600 ms) before discovering services. This settle delay avoids a race on some Android BLE stacks where service discovery immediately after MTU change fails.
5. **Discover services**, locate the service and TX characteristic.
6. **Enable notifications:** set the characteristic notification flag *and* write ENABLE_NOTIFICATION_VALUE (0x01 0x00) to the CCCD descriptor. Data will not flow until the CCCD write completes.
7. You are now connected. Telemetry begins streaming on TX; you may write commands to RX at any time.

iOS / CoreBluetooth note: You do not manually request an MTU on iOS — CoreBluetooth negotiates the largest supported ATT MTU automatically during connection. Steps 2–4 collapse into “connect, then discoverServices”. Enable notifications with `setNotifyValue(true, for:)` on the TX characteristic; CoreBluetooth writes the CCCD for you. Use `.withoutResponse` writes to the RX characteristic and respect `peripheral.maximumWriteValueLength(for: .withoutResponse)` when chunking.

2.4 Writing to the device

Writes to the RX characteristic use **Write Without Response** for throughput. Command strings are short (tens of bytes), so a single write per command is fine. If you ever send a payload larger than the negotiated MTU minus 3 bytes, split it across multiple writes; the device reassembles the byte stream and parses on line terminators, so splitting mid-line is harmless as long as bytes arrive in order.

2.5 Throughput, data rates, and the MTU relationship

This is the key nuance for a robust integration:

- The device’s **default raw mode emits pressure samples at 50 Hz** (one PRS line every 20 ms). At the ASCII line lengths involved, that is a steady, moderate stream that a default 23-byte MTU can technically carry but will fragment heavily.
- **Requesting a large MTU (512) plus a high connection priority is what keeps the 50 Hz stream and IGC bulk transfers smooth.** With a small MTU and default (slow) connection interval you will see backlog, jitter in the vario, and very slow log downloads.
- Notifications can deliver **multiple lines in one packet** and can also **split a single line across two packets**. Never assume one BLE notification equals one line. You must buffer incoming bytes and extract complete lines yourself (section 3).
- Because the stream is continuous and time-sensitive, do all parsing and any onward piping (UI, TCP, file) **off the BLE callback thread**. The reference app pushes every inbound chunk onto a queue consumed by a dedicated foreground-priority worker thread, so a slow UI or disk write can never stall the Bluetooth radio.

3. Message framing

Treat the inbound side as a **byte stream**, not a message stream:

1. Append the bytes from each TX notification to a rolling buffer.
2. Scan the buffer for a line terminator (`\n` **or** `\r`).
3. When found, extract the bytes before it as one line, trim whitespace, and hand it to your parser.
4. Remove the consumed bytes (including the terminator) and continue scanning for the next line.
5. Decode as **US-ASCII**. Do not use UTF-8 multibyte assumptions — the protocol is 7-bit ASCII.

Empty lines are ignored. A partial line at the end of the buffer stays buffered until its terminator arrives in a later notification.

4. Data received from the device

What you receive depends on the device's configured **output mode** (BOM) and **output frequency** (BOF) — see section 5.3 to change them. There are three broad categories of inbound line.

4.1 Output modes overview

Set with `$BOM <n>*`. The full authoritative list (from the Hardware Settings Manual); availability of the higher modes depends on firmware version:

Mode	Name	Line format	Notes
0	BlueFlyVario (default)	PRS <hex>\n	Raw unfiltered pressure, hex Pascals
1	LK8EX1	\$LK8EX1, ...*cc\r\n	Pressure + vario; altitude field sent as 99999
2	LXWP0	\$LXWP0, ...*cc\r\n	Only baro-altitude + vario populated (from filtered pressure at outputQNH)
3	FlyNet	_PRS <hex>\n	Like mode 0 but filtered pressure
4	None	—	Output off (from v9). Useful when using the device purely as a GPS multiplexer
5	BFV custom	\$BFV, ...*cc\r\n	Filtered pressure + vario + temp (+ battery %, + pitot)
6	BFX extended	\$BFX, ...*cc\r\n	As mode 5 plus raw battery voltage
7	OpenVario	OpenVario NMEA	Static pressure derived from Kalman-smoothed altitude (recent v12)
8	BT module debug	—	Bridges USB to the RN4678 module for debugging; not for general use

Which modes the reference Android app parses: 0 (raw), 1 (LK8EX1), 2 (LXWP0), 5 (BFV), 6 (BFX). Modes 3 and 7 are valid device outputs but are not decoded by that app — implement them only if you need them.

BOF (outputFrequency) is a **divisor of the 20 ms base tick**: BOF 1 = every 20 ms (50 Hz), BOF 10 = every 200 ms (5 Hz), BOF 50 = once per second. Valid range 1–50.

Choose based on your app's needs: raw 50 Hz mode (0) if you want to run your own vario filter on unfiltered pressure; a 5 Hz NMEA mode (1, 5 or 6) if you want firmware-computed, Kalman-filtered vario and lower processing load. Most third-party integrations use LK8EX1 (widely supported) or BFV/BFX (richest data).

4.2 Raw key-value Bluefly data telemetry protocol (mode 0)

Plain lines of the form KEY <value>. No checksum. The important ones:

Prefix	Meaning	Encoding	Convert to SI
PRS	Barometric pressure	hexadecimal integer, Pascals	<code>parseInt(hex, 16) → Pa</code>
TMP	Temperature	decimal integer, ×10	<code>value / 10.0 → °C</code>
BAT	Battery voltage	hexadecimal integer, millivolts	<code>parseInt(hex, 16) / 1000.0 → V</code>

Example: PRS 18820 is $0x18820 = 100,384$ Pa. TMP 231 is 23.1 °C. BAT 1068 is $0x1068 = 4200$ mV = 4.20 V.

Pressure at 50 Hz is **raw and noisy**. To display altitude and a usable vario you either (a) select an NMEA mode and let the firmware compute vario, or (b) run your own filter. The reference app uses a Kalman filter on the raw pressure (position-noise ≈ 0.2 , $dt = 0.02$ s) and converts to altitude with the ISA formula in section 4.6.

4.3 NMEA-style Bluefly data telemetry sentences

These follow standard NMEA framing: `$<TYPE>,<fields>*<checksum>`, terminated by CR/LF. The device supports several sentence types so it can drive a wide range of existing flight apps.

\$BFV — BlueFly native (mode 5):

`$BFV,<pressurePa>,<vario_cm/s>,<tempC>,<batteryPct>,<pitotDiffPa>*CC` Pressure is the **filtered** value (unsigned int, Pa); vario is **filtered** (signed int, cm/s — divide by 100 for m/s); temp is signed float °C. Battery % is 0 on models with no battery. The pitot differential-pressure field is only meaningful when the usePitot setting is enabled.

\$BFX — BlueFly extended (mode 6): identical to \$BFV plus raw battery voltage appended: `$BFX,<pressurePa>,<vario_cm/s>,<tempC>,<batteryPct>,<pitotDiffPa>,<batteryVolts>*CC` (The manual notes this sentence is the most likely to evolve as hardware changes.)

\$LK8EX1 — widely supported open standard:

`$LK8EX1,<pressure>,<altitude>,<vario_cm/s>,<tempC>,<batteryVolts>*CC` The BlueFly does **not** populate altitude — it sends 99999 in that field, so ignore it and derive altitude from pressure yourself (section 4.6). Vario is cm/s ($\div 100 \rightarrow$ m/s).

\$LXWP0 — LX-format: `$LXWP0,...,<altitude_m>,<vario_m/s>,...*CC` Here altitude is already in metres and vario already in m/s.

If you only want to support one format, **\$LK8EX1 is the most portable choice** and is understood by most existing free-flight apps. If you want battery voltage as well as percentage, use \$BFX.

4.4 GPS passthrough

On GPS-equipped models the device **multiplexes** GPS NMEA into the same output stream: sentences arriving on the second serial port (UART1) are interleaved with the telemetry output (UART2) at the next 20 ms cycle. From your app's perspective this is transparent — you simply see standard GPS sentences appear inline among the telemetry lines. You will see sentences beginning with \$GP, \$GN, \$GL, or \$GA (e.g. \$GPGGA, \$GPGSA, \$GNRMC). Parse these with any standard NMEA GPS library. In particular:

- ... **GGA** field 6 is the fix quality (0 = no fix).
- ... **GSA** field 2 is the fix type (1 = none, 2 = 2D, 3 = 3D).

The reference app derives a simple GPS status (No message / No fix / 2D / 3D) from these two sentences.

4.5 Checksums

NMEA sentences carry an **8-bit XOR checksum**: XOR of every character between the leading \$ and the *, expressed as two uppercase hex digits after the *.

The reference parser **validates the checksum on the telemetry sentences** (BFV, BFX, LK8EX1, LXWP0) and drops any that fail. GPS and control sentences are accepted without checksum validation in the reference app, but validating everything is good practice. Raw key-value lines (mode 0) have no checksum.

4.6 Converting pressure to altitude

All barometric altitude in the system uses the ISA formula (sea-level reference P0 = 101,325 Pa):

```
altitude_m = 44301.59796 × (1 - (pressurePa / 101325)^0.190295)
```

If you need pressure altitude referenced to a QNH other than standard, either apply your own offset or set the device's BQH (output QNH in Pascals) parameter.

4.7 Control / IGC sentences

Two control sentence types appear on the stream, both related to flight-log handling (\$BFH, \$BFF). These are covered fully in section 6.

5. Commands sent to the device

5.1 Command syntax

Commands are ASCII strings of the form:

```
$<CODE> <argument>*
```

- Start with \$, end with *.

- Query commands take no argument (e.g. \$BST*).
- Set commands take a single argument (e.g. \$BVL 1000*).
- You can concatenate commands: e.g. \$BOM 0*\$BOF 1* sends two in one write.
- **Commands do not require a trailing checksum** — the firmware accepts the bare \$CODE arg* form. (Checksums are only present on the telemetry the device sends back.)

5.2 Reading and writing settings

Settings live in device EEPROM and are exchanged as a **keys line plus a values line**:

1. Send **\$BST*** to request the current configuration.
2. The device replies with three lines:
 - BFV <version...> — firmware version.
 - BST <CODE1> <CODE2> <CODE3> ... — the ordered list of setting **keys** this firmware supports.
 - SET <n> <val1> <val2> <val3> ... — the corresponding **values**, positionally aligned to the keys.
3. To change a setting, send \$<CODE> <hardwareValue>*. After a write, wait briefly (the reference app waits ~300 ms for the EEPROM write) then re-request \$BST* to confirm.
4. **\$RSX*** resets all settings to factory defaults.

Important: the *values* in SET are raw hardware integers. Each parameter carries a **type** and **factor** that define the conversion, and a min/max you must clamp to before writing back:

- **int** — value used directly.
- **boolean** — 0 = false, anything else = true.
- **double** — human value = raw × factor (e.g. factor 100: display = raw ÷ 100 to read, raw = display × 100 to write).
- **int-offset** — human value = raw + factor (e.g. outputQNH, offset 80000).

String parameters (e.g. pilot name) are sent quoted. The full type/factor/range/default for every parameter is in Annex A of the Hardware Settings Manual.

5.3 Output mode and rate

Command	Effect
\$BOM <0-7>*	Set output mode (format family — see 4.1)
\$BOF <1-50>*	Set output rate as a ×20 ms divisor

Common presets used by the reference app's terminal: - **Raw 50 Hz:** \$BOM 0*\$BOF 1* - **LK8EX1 @ 5 Hz:** \$BOM 1*\$BOF 10* - **LXWP0 @ 5 Hz:** \$BOM 2*\$BOF 10*

5.4 Other serial commands

Beyond settings, the firmware accepts a number of action commands (all \$. . . * framed):

Command	Effect
\$TMP*	Query temperature; device replies TMP <°C×10>
\$RST*	Hot reboot (from v9)
\$RSX*	Reboot and reset all settings to defaults (factory reset, from v9)
\$SLP*	Sleep; wake by sending any character on the port
\$SLX*	Sleep; can only be woken by a power cycle
\$BTN*	Simulate a hardware button press (toggles audio, etc.)
\$BVU*	Double the current volume (clamped to 1000) — short confirmation beep
\$BVD*	Halve the current volume (clamped to 1) — short confirmation beep
\$BSD <freq> <ms> . . . *	Play a tone (or a queued sequence of freq/duration pairs). e.g. \$BSD 400 100* = 400 Hz for 100 ms. Whole command ≤ 82 chars
\$RNC <cmd>*	Pass a command through to the RN4677/RN4678 Bluetooth module (advanced; restarts the module)

\$BSD is handy for custom audio feedback (confirmation chirps, alerts) driven from your app without touching the vario's own tone logic.

5.5 Settings parameter reference (selected)

The firmware exposes 40+ tunable parameters. Codes are stable; availability depends on firmware version (the BST keys line tells you which are supported). A representative subset:

Code	Name	Type	Scale	Notes
BVL	volume	double	÷1000	non-linear; range varies by firmware
BFL / BOL	lift start / stop threshold	double	÷100	m/s
BFS / BOS	sink start / stop threshold	double	÷100	m/s
BFQ / BFI	lift base freq / increment	int	—	Hz
BSQ / BSI	sink base freq / increment	int	—	Hz
BRM	rate multiplier	double	÷100	beep cadence
BAM	audio mode	int	—	0 = tone, 1 = vibration

Code	Name	Type	Scale	Notes
BQH	output QNH	int- offset	+80000	Pascals (stored value = QNH – 80000)
BOM	output mode	int	—	0–7 (see 4.1)
BOF	output frequency	int	—	×20 ms divisor
BUP	use pitot (airspeed)	bool	—	MS4525DO sensor
BGL	GPS log interval	int	—	seconds
BPN	pilot name	string	—	quoted

Always drive your UI from the live BST/SET exchange rather than hard-coding the full list, so your app adapts to whatever firmware the connected device runs.

6. IGC flight-log download protocol

The BlueFly stores flights internally as IGC files and streams them out over the same BLE link on request. The flow is a small command/response state machine.

6.1 Listing available flights

1. Send **\$BLF*** to list flights.
2. The device replies with a series of **\$BFH, ..., <startSector>, <fileName>*CC** header sentences — one per stored sector. Build your file list by grouping headers by filename:
 - First header for a filename → create a file entry with its startSector, sector count 1.
 - Subsequent headers for the same filename → increment its sector count.
 - If a later header for the same filename reports a *different* start sector, flag it as an error (partial/corrupt data).

Filenames use the pattern **BFVddMMyy_HHmms.igc** where the timestamp is **UTC**. Parse it to show local start time.

Sector maths (useful for progress bars and duration estimates before download):

```
usableBytes = (sectorCount - 1) × 4064 + 2032
durationSecs = usableBytes / 37           (≈ 37 logged bytes per second)
sizeKB       = usableBytes / 1024
```

6.2 Downloading a flight

1. Open your output stream/file **first**.
2. Send **\$BFF <startSector>*** (the start sector of the chosen file).
3. The device sends **\$BFF,1*** — the “transfer starting” signal. **Do not write any data to your file until you receive this.** Everything before it is protocol chatter; the **\$BFF,1*** opens the gate.

4. The device then streams the **raw IGC content as ordinary text lines** (lines that do **not** begin with \$). While a download is in progress, treat every non-\$ line as file content: append it plus `\r\n` to your output and update your byte counter for progress.
5. The device sends **\$BFF,2,<byteSize>*** — the “transfer finished” signal, with the total size. Flush and close your file.

State discipline is important: keep a single “downloading” flag. Set it on **\$BFF,1***, clear it on **\$BFF,2,...***. Only capture non-command lines while it is set.

6.3 Erasing / formatting log memory

Send **\$BXF*** to format (erase all flights) and restart the device. This can take up to ~100 seconds; the device will disconnect and re-advertise. Warn the user and clear your local file list.

7. Recommended integration workflow

A minimal, robust client does the following:

1. **Scan** (filter by name or service UUID), let the user pick a device, stop the scan.
2. **Connect** and run the exact sequence in section 2.3 (large MTU, high priority, settle, discover, enable notifications).
3. **Buffer and line-split** the TX stream (section 3); parse on a worker thread, never the BLE callback thread.
4. **Select an output mode** to match your app — e.g. **\$BOM 1*\$BOF 10*** for LK8EX1 at 5 Hz — and parse the resulting telemetry (section 4.3).
5. **Optionally** query settings with **\$BST*** and expose the parameters you care about.
6. **Optionally** list (**\$BLF***) and download (**\$BFF***) flight logs following the state machine in section 6.
7. On disconnect, tear down cleanly and (if desired) auto-reconnect using the last MAC/identifier.

8. iOS-specific notes (CoreBluetooth)

- **Identify by peripheral, not MAC.** iOS hides the hardware MAC; persist the `CBPeripheral.identifier` (a per-app UUID) instead, and use `retrievePeripherals(withIdentifiers:)` to reconnect.
- **No manual MTU.** ATT MTU is negotiated automatically. Skip steps 2–4 of section 2.3 and just `discoverServices` after connecting.
- **Notifications:** `setNotifyValue(true, for: txCharacteristic);` CoreBluetooth handles the CCCD.
- **Writes:** use `.withoutResponse;` check `maximumWriteValueLength(for: .withoutResponse)` and chunk anything larger.

- **Background:** declare the bluetooth-central background mode if you need to stay connected while backgrounded. iOS state-preservation/restoration will reconnect known peripherals.
- Everything in sections 3–6 (the ASCII protocol) is identical on iOS — only the BLE plumbing differs.

9. Reliability and background operation

Lessons baked into the reference app that are worth replicating:

- **Keep the radio thread clear.** Queue inbound chunks and process them on a dedicated worker; pipe to UI/network/disk from that worker, not from the BLE callback.
- **Hold a wake lock / background mode** during flight so the OS does not suspend the connection. On Android, run a foreground service of type `connectedDevice` and prompt the user to exempt the app from battery optimisation. On iOS, use the bluetooth-central background mode.
- **autoConnect = true** (Android) helps ride out brief signal dropouts common in flight.
- **Expect reconnects after \$BXF*** (format) and after firmware operations — the device disconnects and re-advertises.
- **Permissions (Android 12+):** `BLUETOOTH_SCAN`, `BLUETOOTH_CONNECT`, and location (`ACCESS_FINE_LOCATION`) — the latter is still required here because the app also consumes GPS NMEA, so you cannot use the `neverForLocation` scan flag. Android 13+ also needs `POST_NOTIFICATIONS`; Android 14+ needs `FOREGROUND_SERVICE_CONNECTED_DEVICE`.

10. Quick reference

Inbound line prefixes

Prefix	Category	Section
PRS / TMP / BAT	Raw telemetry (mode 0)	4.2
\$BFV, / \$BFX,	BlueFly NMEA telemetry	4.3
\$LK8EX1, / \$LXWP0,	Standard NMEA telemetry	4.3
\$GP... \$GN... \$GL... \$GA...	GPS passthrough	4.4
BFV / BST / SET	Settings response	5.2
\$BFH,	IGC file header (listing)	6.1
\$BFF,1* / \$BFF,2,...*	IGC transfer start / end	6.2

Outbound commands

Command	Purpose	Section
\$BST*	Request settings (keys + values)	5.2
\$<CODE> <value>*	Set a parameter	5.2
\$RSX*	Factory-reset settings + reboot	5.2 / 5.4
\$RST*	Hot reboot	5.4
\$BOM <0-7>*	Set output mode	4.1 / 5.3
\$BOF <1-50>*	Set output rate (×20 ms)	5.3
\$TMP*	Query temperature	5.4
\$BSD <freq> <ms> ...*	Play tone / tone sequence	5.4
\$BVU* / \$BVD*	Double / halve volume	5.4
\$BTN*	Simulate button press	5.4
\$SLP* / \$SLX*	Sleep (wakeable / power-cycle-only)	5.4
\$BLF*	List IGC flight logs	6.1
\$BFF <startSector>*	Download a flight	6.2
\$BXF*	Format (erase) log memory	6.3

Key constants

Constant	Value
Service UUID	49535343-FE7D-4AE5-8FA9-9FAFD205E455
TX (notify)	49535343-1E4D-4BD9-BA61-23C647249616
RX (write no-response)	49535343-8841-43F4-A8D4-ECBE34729BB3
Requested MTU	512
Base output tick	20 ms (50 Hz)
Sea-level pressure P ₀	101,325 Pa
Altitude formula	$44301.59796 \times (1 - (P/P_0)^{0.190295})$
IGC filename pattern	BFVddMMyy_HHmms.igc (UTC)