HoloQ Player - Serial Protocol Integration Guide

HoloQ Technologies

2026

Contents

HoloQ Player — Serial / Protocol Integration Guide

Version: 1.1 Audience: developers implementing the **other end** of the serial link — a PC, an Arduino, a Raspberry Pi, or any embedded controller capable of raw serial I/O. Nothing in this document assumes the other end is a PC specifically; "the controller" is used throughout to mean whatever device you are building. Covers the current release, including the start-signature handshake required for reliable link bring-up (§3.1).

---

1. Overview

HoloQ Player communicates with an external controller over a single serial link using a **single-byte, unframed protocol** — every command and every event is exactly one byte, sent and received with no header, no length field, and no checksum. This document specifies that byte-level contract completely: what the Player sends, what it listens for, and the timing/ordering guarantees (and non-guarantees) around each.

One optional exception to "no acknowledgment": a Player can be configured to require the controller to identify itself with a start signature before it will act on anything, which it acknowledges with a single byte. This link bring-up handshake is **off by default** — most deployments, including any with no external controller at all, never encounter it — but if you're integrating a controller and the device you're targeting has it enabled, every other command in this document is gated behind it. See §3.1 to check whether it applies to your deployment.

This document is self-contained — a controller can be implemented from this document alone, without needing the Player's own source or the deployment config/playlist specs.

---

2. Physical / Link Layer

Configured on the Player's side via `config.json`'s `serial` block (see the Configuration Specification for the full field reference). The controller must match these settings exactly:

Parameter	Player default	Notes
Baud rate	9600	Must match exactly
Data bits	8	5–8 supported by the Player
Parity	None	Even/Odd also supported
Stop bits	1	1 or 2
Mode	Binary	Raw byte values, not ASCII text encoding
Logic	Normal	Inversion available if your link requires it

All of these are configurable per-deployment on the Player's side — confirm the actual values in use for the specific device you're integrating with, rather than assuming the defaults.

---

3. Protocol Fundamentals

- **Every message is exactly one byte**, value range 0–255 — with the single exception of the start signature used during link bring-up (§3.1), which is a short multi-byte string.
- **No framing, and at most one acknowledgment in the entire protocol** — the handshake ack (§3.1, §4.4), which only exists at all if the handshake is enabled for the deployment you're integrating with (it is **not** by default). Nothing else the controller sends gets a reply; the Player does not confirm receipt of jump triggers, transport commands, or anything else.
- **Two independent byte namespaces, by direction:**
  - **Player → controller** ("send" bytes): heartbeat, item lifecycle events, timed events, and — only if the handshake is enabled — the handshake ack. These are informational; the Player does not wait for any response.
  - **Controller → Player** ("receive" bytes): jump triggers and global commands (reboot, transport, speed). By default the Player acts on these immediately on receipt. **If the handshake is enabled** (§3.1), it instead acts on them only after link bring-up has completed — before that, every one of these bytes is silently discarded.
- **Byte values are not inherently unique across categories.** A given numeric value might be a legitimate "send" byte in one deployment's playlist and coincidentally the same numeric value as an unrelated "receive" byte in `config.json` — direction is what disambiguates them, not the value itself. (The Player does validate that a playlist's jump bytes don't collide with the reserved global receive-codes — see §6 — but this is a load-time playlist validation, not a protocol-level framing distinction.)

3.1 Link bring-up (start-signature handshake)

This is an optional feature, off by default. Most deployments — anything auto-playing on a loop with no external controller, and plenty of controller-driven deployments too — never need it and never encounter it. It exists specifically for deployments with an external controller where protection against spurious commands during power-up matters enough to be worth the extra integration step. **Check `config.json: serial.handshake` on the specific device you're targeting before assuming this section applies** — if it's `false` (the default), skip straight to §4; every command in this document works immediately with no signature required.

When enabled, the Player ignores every received byte — reboot included — until the controller has identified itself. This exists because the serial link carries no framing, addressing, or checksum: without it, the Player can't tell a deliberate command from whatever else happens to be on the wire — a floating RX line, the

break a PC's USB-serial adapter emits when it opens the port, or the burst a controller's own bootloader transmits on its way up. Any of those can happen to match a valid command byte, producing unexplained playback jumps or spurious reboots around start-up in a deployment that hasn't turned this protection on.

Configuration — all of the following live under the `serial` object in `config.json` and are fully overridable per deployment. The values below are defaults that apply when the key is absent, not fixed protocol constants — always confirm the actual values for the device you're integrating with.

Behavior	Default	Configured where
Handshake required	**disabled**	`config.json: serial.handshake`
Start signature	`"HOLOQSTART"`	`config.json: serial.start_signature`
Handshake ack byte	246	`config.json: serial.handshake_ack_send`
Max gap between signature bytes	2000 ms	`config.json: serial.signature_gap_ms`

How it works, step by step (once `serial.handshake: true` is set):

1. **Wait for the Player to be alive.** The recommended pattern is to wait for three consecutive heartbeats (§5.1), spaced at roughly the configured heartbeat interval, before announcing yourself. Nothing on the Player enforces this — it's a controller-side convention — but sending the signature before the Player's serial reader is up just means it's missed, and you fall back to the retry step below anyway.
2. **Send the start signature in a single write**, so its bytes are contiguous on the wire. Matching is exact and case-sensitive, and **any non-matching byte in the middle resets the match to the start**. Consecutive signature bytes must also arrive within `serial.signature_gap_ms` of each other, or the partial match resets (a value of `0` disables this timing check entirely).
3. **Wait for the ack byte** (§4.4). If none arrives within about 2 seconds, resend the signature, and keep resending indefinitely until it arrives. This retry loop matters: without it, a single corrupted signature leaves the Player permanently deaf while the controller believes it's ready.
4. **Once acknowledged, every other command in this document works normally.** Sending the signature again mid-session — for example, because the controller itself restarted — is harmless: the Player re-acknowledges and playback continues undisturbed.

What is *not* gated: everything the Player sends — heartbeat, item lifecycle events, timed events — flows normally from the start of playback, before and during the handshake. This is deliberate, not an oversight: the controller is waiting for heartbeats before it even sends the signature (step 1 above), so gating the heartbeat too would deadlock the handshake permanently.

A signature match is the only real proof your controller can transmit at all. The heartbeat only proves the Player-to-controller direction works; a controller with a broken TX line but a working RX line would otherwise look perfectly healthy right up until you send a real command and nothing happens.

Known limitation — mid-session controller reset. The handshake relies on ordering: on a cold start, a controller's power-up noise is emitted *before* its firmware runs and sends the real signature, so that noise lands while the Player is still unready and gets discarded harmlessly. That ordering only holds on a cold start. If the controller resets while the Player is **already** past bring-up, its power-up noise now lands on an open, listening link — and the signature that would normally flag the restart arrives *after* the noise, not before it. Two deployment-level mitigations, both outside the Player itself:

- Move `reboot_receive` off `255` — an undriven RX line idles at `0xFF`, and that value reboots the device by default.
- If the controller is an Arduino, avoid pins 0/1 for this link — that's the bootloader's own UART, which transmits for roughly a second after every reset.

Do not treat the handshake as covering every possible source of startup noise — it solves the cold-start case completely; a mid-session controller reset needs one of the mitigations above as well.

---

4. What the Player Sends (Player → Controller)

Two categories: a fixed set of **global** signals (heartbeat), and per-playlist **item-defined** signals (lifecycle events, timed events). The global signal's byte value is fixed per-device via `config.json`; the item-defined signals' byte values are chosen by whoever authored that specific playlist and can differ between playlists and even between items within the same playlist.

4.1 Heartbeat

Sent periodically while the Player considers itself healthy. See §5 for the full liveness model — this is the single most important signal in the protocol for a controller to monitor.

4.2 Item lifecycle events (`on_start_send` / `on_end_send`)

- `on_start_send` fires once, the instant an item becomes current (whether reached via natural advance or a jump).
- `on_end_send` fires once, when an item finishes and the Player is about to advance to its `next` item.
- **Neither field is required to exist on any given item.** A playlist author may omit either or both. If both are omitted, that item's start/end are entirely invisible to the controller — there is no fallback signal.
- `on_end_send` does **not** distinguish forward-finish from reverse-finish — see §9.3.

4.3 Timed events (`time_triggers[]`)

Each entry fires once, when playback of the *current* item crosses a specified elapsed-time threshold (`elapsed_ms`, measured from that item's own start).

Critical exception — read this before assuming any `time_triggers` will fire: *any item that has a `reversed_file` set — i.e. any reverse-paired item — has its `time_triggers` permanently ignored, unconditionally, regardless of whether that item is ever actually played in reverse.* This is not a bug; it is deliberate Player behavior (rationale: once an item can switch direction, "elapsed time since start" stops being a well-defined concept for it). If a playlist assigns `time_triggers` to a reverse-paired item, those bytes will simply never arrive — see the worked example in §11 for exactly what this looks like from the wire.

`time_triggers` only apply to video items — never to images.

4.4 Handshake acknowledgment (`handshake_ack_send`)

Sent exactly once, the moment the Player matches the controller's start signature (§3.1) — this is the one and only reply the protocol defines. Default byte `246`, configured via `config.json: serial.handshake_ack_send`.

If the controller announces itself again mid-session (for example, after its own restart), the Player sends this same byte again as a re-acknowledgment; playback is not affected either way.

This byte is reserved: no playlist item may use it as an `on_start_send`, `on_end_send`, or `time_triggers[].send` value — see §6 and the Playlist Specification for the validation rule.

---

5. Heartbeat & Liveness Model

This is the model a controller's own recovery logic needs to be built around — the Player deliberately implements none of this itself.

5.1 Normal operation

While playback is healthy, the Player sends its heartbeat byte approximately every `watchdog_interval_ms` (device-configured, default 5000ms).

5.2 Stall behavior

If playback makes no progress for `stall_detect_ms` (device-configured, default 5000ms) after having started, the Player **stops sending the heartbeat and does nothing else** — no retry, no auto-restart, no recovery attempt of any kind. This is intentional. **The absence of a heartbeat is the only signal a controller gets that something is wrong** — the Player will never distinguish "I stalled," "I'm hung," or "I crashed" over the wire; from the controller's perspective they all look identical: silence.

5.3 What the Player will NOT do for you

- It will not retry a stalled item.
- It will not restart itself.
- It will not attempt any self-recovery of any kind.

Recovery is entirely the controller's responsibility. This was a deliberate design decision, not a limitation to work around — build your controller's logic expecting to own this.

5.4 Recommended controller-side pattern

```
heartbeat missed for > stall_detect_ms + margin
  -> do not act immediately (see 5.5's grace-period note)
  -> wait a reasonable grace period (recommendation: 40-60s, based on
     empirically observed device reboot times of ~30s when a hardware
     reset mechanism is engaged)
  -> did the heartbeat resume on its own within that window?
       YES -> log as a transient/self-recovered event, no action needed
       NO  -> escalate: attempt reboot_receive (works only if the process
              is hung-but-alive, not if it's genuinely dead - see 5.5),
              then fall back to whatever external hardware reset mechanism
              your deployment provides (see 5.6)
```

A resumed heartbeat is not, by itself, a resumed link — if the handshake is enabled for this deployment. Whenever a heartbeat gap exceeds `stall_detect_ms` plus your margin — whether it self-recovered or you had to intervene — and `serial.handshake` is `true` on this device, treat the handshake state as invalid and re-run the full bring-up sequence from §3.1 (wait for heartbeats, send the signature, wait for the ack) **before sending any other command.** A rebooted or reset Player has forgotten it was ever acknowledged; commands sent on the old assumption that the link is "ready" will simply be discarded until the signature is sent again. If the handshake is disabled (the default), this step doesn't apply — simply resume sending commands once the heartbeat returns.

5.5 `reboot_receive`'s real scope — do not over-rely on it

`reboot_receive` reaches the Player's serial-reader thread directly, independent of the main/UI thread — meaning it works even if the Player's main thread is completely hung (an ANR-equivalent state). **But if the Player's entire process has actually crashed, `reboot_receive` does nothing** — there is no code

running anywhere in the process to receive it. A controller cannot distinguish “hung” from “crashed” from the wire (both just look like a missing heartbeat), so **always plan a fallback beyond `reboot_receive`** for the case where it silently has no effect.

5.6 Recovering a genuinely dead device

Since `reboot_receive` cannot help once the process is truly dead, a production deployment needs an independent, physical recovery path — this is necessarily outside the serial protocol itself (by definition: the serial link depends on the same process that just died). The approach adopted for this project is an external relay/microcontroller, on a **separate** control channel from the playback serial link, that briefly shorts the board’s physical reset-button contacts on command — electrically identical to a person pressing the button, and therefore effective regardless of what state the Player’s software is in. Designing and wiring this is specific to your deployment’s hardware and outside the scope of this protocol document, but the principle worth carrying into your controller’s design is: **build a recovery path that does not depend on the same link you’re trying to recover.**

---

6. What the Player Listens For — Global Commands (Controller → Player)

If `serial.handshake` is enabled on this device, none of these work until link bring-up (§3.1) has completed — `reboot_receive` included, with no exceptions. A byte matching one of these codes, sent before the Player has acknowledged your start signature, is silently discarded, not queued for later. If a command appears to have no effect on a device with the handshake enabled, confirm it actually completed before assuming anything else is wrong. **On the default configuration (handshake disabled), every command below works immediately** — this gating simply doesn’t apply.

These 8 codes are configured per-device in `config.json` and are checked **before** any playlist-defined jump trigger — a global command can never be shadowed by a playlist’s own `serial_triggers`. **Current defaults** — always confirm against the actual deployed device’s `config.json`, since every one of these is fully overridable:

Command	Byte (default)	Effect
`reboot_receive`	255	Reboot the device (see §5.5 for its real scope)
`stop_receive`	254	Freeze the current frame
`play_right_receive`	253	Play forward — also the “resume” command after a `stop`
`play_left_receive`	252	Play in reverse — **only meaningful on reverse-paired items** (§9); ignored otherwise
`speed_fast_receive`	251	Set playback speed to 2.0×
`speed_regular_receive`	250	Set playback speed to 1.0×
`speed_slow_receive`	249	Set playback speed to 0.5×
`speed_ultraslow_receive`	248	Set playback speed to 0.25×

Note on history: a `hot_reload_receive` command existed in earlier revisions of this protocol and has been **completely removed**. It was documented as reloading the playlist without a reboot; it was never actually implemented in the Player despite being documented, and the field no longer exists in `config.json` at all. If you’re integrating against older documentation or example code, discard any reference to this command — there is no live-reload mechanism in the current protocol; a playlist or config change always requires a full reboot to take effect.

6.1 Playlist-defined jump triggers (`serial_triggers`)

Beyond the 8 global codes, each playlist item can define its own `serial_triggers` — arbitrary byte values (chosen by whoever authored that playlist, not fixed by this protocol) that, while that item is current, cause an **immediate, seamless jump** to a named target item. These values are validated at playlist-load time to never collide with the 8 reserved global codes above (a playlist that violates this is refused entirely by the Player, not silently partially honored) — but you must still obtain the actual jump-byte values for a specific deployment from that deployment's playlist file; they are not fixed by this protocol document.

---

7. Ordering & Timing Guarantees

- **Jumps are immediate and seamless** — expect the target item's `on_start_send` (if defined) to arrive promptly after sending its jump byte, with tight timing (well under a second under normal conditions).
- **`time_triggers` fire once each**, in whatever order their `elapsed_ms` values dictate, always before that item's `on_end_send` (assuming the item's own real playback duration exceeds the largest `elapsed_ms` value defined on it — this is a property of the actual media file, not something the protocol enforces).
- **On entering any item** (whether via natural advance or a jump): its `on_start_send` fires (if defined), its speed resets to 1.0×, and its direction resets to forward — regardless of what speed/direction was active on the previous item.

---

8. Reverse-Paired Items — Transport & Speed Behavior

An item is "reverse-paired" if its playlist entry has a `reversed_file` field. Only reverse-paired items respond meaningfully to `play_left_receive`; sending it to a non-reverse-paired item has no effect.

8.1 The position-mapping model

When `play_left_receive` is sent at elapsed position `t` (time since the current item started), the Player switches to the item's `reversed_file`, starting playback from the **mapped position** `D − t` (where `D` is the item's total forward duration). Because the reversed file's own remaining duration from that mapped position is `D − (D − t) = t`, **the item will take exactly `t` more time to reach its end** — not `D − t`, which is what continuing to play forward would have taken instead.

This gives a controller a reliable way to verify a reverse command actually took effect, purely from timing: measure the time from sending `play_left_receive` to that item's next `on_end_send`. If it's close to `t` (the position at the moment you sent the command), reverse engaged correctly. If it's close to `D − t` instead, the command had no effect (the item just kept playing forward) — see §11.2 for a worked example of this exact check.

8.2 `on_end_send` does not indicate direction

Whether an item finishes by reaching the natural end of its forward file or the natural end of its reversed file, the same `on_end_send` fires either way (if defined at all). Direction must be inferred from timing (§8.1), not from which event fired.

8.3 Muting

Reverse-paired items play muted, regardless of direction, whenever a reversed_file is present (this is unconditional, not toggled by whether reverse is currently engaged).

---

9. Worked Examples

9.1 A full natural-playback trace

Given a playlist where `clip1.mp4` (`on_start_send=1`, `on_end_send=4`, `time_triggers` at 2000ms→2 and 4000ms→3) is followed via `next` by `clip6.mp4` (`on_start_send=5`, `on_end_send=7`, one `time_trigger` at 1500ms→6), then `clip3.mp4` (`on_start_send=8`, `on_end_send=10`, one `time_trigger` at 1000ms→9), then `clip7-4k.mp4` (`on_start_send=11`, `on_end_send=14`, `time_triggers` at 100ms→12 and 3000ms→13) — **with no `serial_triggers` sent during the observation window** — the expected byte sequence over serial (excluding heartbeat) is:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
```

9.2 The same playlist, but three of those items are reverse-paired

If `clip1.mp4`, `clip6.mp4`, and `clip3.mp4` (but *not* `clip7-4k.mp4`) each also have a `reversed_file` set, the `time_triggers` on those three items (2, 3, 6, 9) are **permanently skipped** per §4.3/§8 — they will never arrive, regardless of whether reverse mode is ever engaged on those items. Only `clip7-4k.mp4` (the sole non-reverse-paired item) fires its `time_triggers` (12, 13) normally. The actual observed sequence becomes:

```
1, 4, 5, 7, 8, 10, 11, 12, 13, 14
```

This is not a dropped-byte error and not a controller-side reception bug — it is the Player behaving exactly as specified once you account for which items are reverse-paired. Before treating a "missing" byte as a fault, always check whether the source item has `reversed_file` set.

9.3 Verifying a `play_left` command actually engaged

For a reverse-paired item with a known baseline duration `D` (measured on a prior, uncommanded pass through the item):

```
1. Note on_start_send timestamp -> t_start
2. Send play_left_receive at some elapsed time t (t_switch - t_start)
3. Note on_end_send timestamp -> t_end
4. actual_remaining = t_end - t_switch

   actual_remaining ~= t              -> reverse engaged correctly
   actual_remaining ~= (D - t)        -> command had NO effect (still
                                         forward) - possible causes include
                                         a missing/mismatched reversed_file
                                         on that item, or the command byte
                                         not actually reaching the Player
```

---

10. Appendix — Full Byte Map Summary

Direction	Purpose	Byte (default)	Configured where
controller → Player	Start signature (bring-up)	`"HOLOQSTART"` (multi-byte string, not a single value)	`config.json: serial.start_signature`
Player → controller	Handshake ack	246	`config.json: serial.handshake_ack_send`

Direction	Purpose	Byte (default)	Configured where
Player → controller	Heartbeat	247	`config.json: watchdog_heartbeat_send`
Player → controller	Item start	per-item, playlist-defined	`playlist.hlq: on_start_send`
Player → controller	Item end	per-item, playlist-defined	`playlist.hlq: on_end_send`
Player → controller	Timed event	per-item, playlist-defined	`playlist.hlq: time_triggers[].send`
controller → Player	Jump	per-item, playlist-defined	`playlist.hlq: serial_triggers[].receive`
controller → Player	Reboot	255	`config.json: reboot_receive`
controller → Player	Stop (freeze)	254	`config.json: stop_receive`
controller → Player	Play forward / resume	253	`config.json: play_right_receive`
controller → Player	Play reverse	252	`config.json: play_left_receive`
controller → Player	Speed 2.0×	251	`config.json: speed_fast_receive`
controller → Player	Speed 1.0×	250	`config.json: speed_regular_receive`
controller → Player	Speed 0.5×	249	`config.json: speed_slow_receive`
controller → Player	Speed 0.25×	248	`config.json: speed_ultraslow_receive`

All single-byte values are 0–255; only the low byte of any value is meaningful on the wire. The start signature is the one exception — a short printable-ASCII string, matched byte-by-byte rather than as a single value. Every default above is overridable per-device — confirm against the actual deployed `config.json` before integrating against a specific device. Both `handshake_ack_send` and `watchdog_heartbeat_send` are reserved: no playlist item's `on_start_send`, `on_end_send`, or `time_triggers[].send` may equal either configured value — a playlist that does is refused entirely at load time.

11. Glossary

- **Link bring-up / handshake** — the start-signature exchange (§3.1) that must complete before the Player will act on any other command.
- **Start signature** — the string (default `"HOLOQSTART"`) a controller sends once to identify itself and unlock the rest of the protocol.
- **Handshake ack** — the single byte (default 246) the Player sends back the moment it matches the start signature.
- **Reverse-paired item** — a playlist item with `reversed_file` set, enabling `play_left_receive` on it and unconditionally disabling its `time_triggers`.
- **Global command** — one of the 8 fixed `config.json`-configured receive-codes (§6), always checked ahead of any playlist-defined jump.
- **Jump trigger** — a playlist-defined `serial_triggers` entry; a receive byte that causes an immediate jump to a named item, scoped to whichever item is currently active.

- **Elapsed time (for `time_triggers`)** — measured from the current item's own `on_start_send` moment, not from device boot or any other reference point.