Distributed Real-Time Servo Control

Author Maximilian Ardalan Bernhard

Institutions FH Joanneum, KUG

Course Design & Research 2

Date 2026-08-31

1. How the Movement Is Made

Before the two boards can follow each other, I had to decide what kind of movement the installation should produce. The speaker gimbal is the source of that motion, and I chose to let it wander rather than trace a fixed path. Its movement wanders from one aim point to the next: every so often it picks a new pan and tilt at random within the servo limits, holds that point for a random interval, and then moves on. The dwell time is drawn at random from a range of one and a half to five seconds, so the pacing stays irregular rather than metronomic. The aim points are produced by the Arduino `random()` function, which is not seeded, so the controller starts from the same pseudo-random sequence on every boot. That choice was deliberate, because it makes the motion reproducible during testing.

Raw jumps between aim points would make the gimbals snap, so each new target is approached through a ramp. A dedicated servo task moves the current angle toward the target by a fixed step every tick, at most one degree per fifty-millisecond tick. The result is a smooth, linear sweep between targets: the pan and tilt glide between points instead of jumping. This is the smooth, ramped movement data that the master then shares with the follower.

2. The Master/Follower Principle

With the movement defined, the two boards take distinct roles:

- The `SpeakerGimbal` acts as the master: it generates the aim points, moves its own servos, and broadcasts the result to the follower.
- The `MicGimbal` acts as the follower: it receives the master's aim, applies a correction, and moves its own servos to match.

Both devices stand in front of the same wall at the same height, facing a common area of that wall.

Because they stand at the same height, tilting is passed through directly. The master and the follower aim at the same vertical angle, so the follower adopts the master's tilt unchanged.

Panning, however, needs a geometric correction. The master and the follower are horizontally offset from each other, so to hit the same point on the wall they must aim at two different horizontal angles. If the master points straight at a spot on the wall, the follower, standing a step to the side, must aim across that offset to reach the same spot. The corrected pan is given by $f_{\text{pan}} = \operatorname{atan}\left(\tan\left(m_{\text{pan}}\right) + \frac{\Delta_x}{d}\right)$, where m_{pan} is the master's pan, Δ_x is the horizontal separation of the two gimbals, and d is each gimbal's distance to the wall. The formula maps the master's angle onto a lateral position on the wall, then aims the follower at that position from its own offset point. This is a simplified case of parallax correction. A fuller implementation could also correct for a vertical offset between the boards and for walls that are not flat, but neither was needed in this prototyping phase.

3. Bluetooth Debugging

Getting the control channel to behave turned into the most frustrating part of the project. My first attempt carried the pan and tilt data over SPP (the Bluetooth Serial Port Profile), instead of BLE, alongside the A2DP audio link. SPP is the classic Bluetooth profile that emulates a serial connection, and it seemed the natural way to send short control messages.

SPP failed silently at first. Nothing appeared to happen, and there was no useful error to chase. Print debugging finally revealed the cause: two Bluetooth libraries were both trying to initialize the Bluetooth controller, and the second initialization was silently failing. Once I fixed the order of initialization, the control data began to flow.

But making SPP work seemed to disturbed the audio. As SPP runs on the same Classic Bluetooth radio as A2DP, the two protocols competed for the same bandwidth. The added radio traffic starved the audio stream, and the sound doubled and stuttered, like a buffer repeating until a new one arrived. This was worse than the original problem, so I made a decision: fix the audio first, and reimplement the control transport afterwards.

With the audio as the priority, I looked for an alternative for the control channel. ESP-NOW was tempting, because it is simple and direct. It was not a real option here, though, because ESP-NOW runs on WiFi, and on the ESP32 the WiFi and Bluetooth

share the same 2.4 GHz radio; the two cannot operate at full speed at the same time, and combining ESP-NOW with A2DP leads to audio dropouts (Schatzmann 2026).

The answer was Bluetooth Low Energy. BLE is a separate protocol stack that runs inside the same Bluetooth controller as A2DP, so the two coexist cleanly when the controller is set to dual mode. Moving the control traffic onto BLE solved the stutter without sacrificing the servo link, and the final architecture rests on that coexistence.

4. Bluetooth Dual-Mode Architecture

The final architecture brings audio, control data, and motion together inside one device. Both protocols live in the same Bluetooth controller, and making them coexist came down to two details: the controller's mode, and the order of initialization. The controller must run in dual mode, ESP_BT_MODE_BTDM, so that it can carry the Classic Bluetooth audio and the Low Energy control data at once. The mode is set before the audio stack starts, because the A2DP library initializes the controller in that mode. BLE is initialized after the audio, and its init call detects that the controller is already running and registers the control service on top of it. This ordering is the same fix that the earlier silent failure came down to.

The control channel itself is a custom GATT service. Both boards use the same custom 128-bit UUIDs for the service and its characteristic, so each side recognizes the other's data. The master broadcasts its current pan and tilt as a packed structure of two 16-bit integers, four bytes in total. It sends this as a notification ten times per second, which is fast enough for the slow gimbal motion while keeping the radio airtime small.

The packed structure replaced an earlier text format. The old messages looked like "P,90,90" and took about nine bytes, and the receiver had to parse them. The packed structure is four bytes, needs no parsing on the receiver, and fits easily within the default Bluetooth Low Energy MTU of 23 bytes. Choosing a compact binary layout made the link simpler and faster.

The motion logic runs on a dedicated FreeRTOS task, pinned to the second core at low priority and ticking every fifty milliseconds ("FreeRTOS Documentation" 2026). Keeping this task off the main loop stops it from disrupting the timing-sensitive Bluetooth tasks,

which had caused audio dropouts. The shared state between the Bluetooth callback and the servo task is small: a pair of float arrays holding the current and target angles. Four-byte-aligned floats load and store atomically on the Xtensa LX6, so the two tasks share that state without a mutex. The compact wire format and the atomic shared floats solve two different problems. The wire format saves radio bytes, while the atomic floats let the tasks share state without locking.

Bibliography

"FreeRTOS Documentation." 2026. https://freertos.org/Documentation/00-Overview.

Schatzmann, Phil. 2026. "ESP32-A2DP.". https://github.com/pschatzmann/ESP32-A2DP.