ChatGPT

Prioritized Tasks for Upgrading fyp-gsr-windows (Synchronization with Android)

Below is a comprehensive list of implementation tasks for improving **fyp-gsr-windows**, prioritized from critical synchronization fixes to lower-priority enhancements. These tasks focus on robust timing alignment with the **fyp-gsr-android** app and ensuring smooth cross-platform data capture.

Critical Synchronization Fixes and Improvements

1. **Implement a Robust Cross-Device Time Sync Protocol:** Establish a master clock reference between the PC and Android. For example, have the PC send a `sync_time` command with its timestamp so phones can calibrate their local monotonic clocks to the PC's timeline [1] [2]. This network time calibration will unify timestamps across devices, aiming for sub-millisecond alignment as advertised [3]. Use Android's existing monotonic clock base and adjust for the offset when the sync command is received, so all recorded data events share a common time origin.

2. **Calibrate and Log Initial Clock Offset:** When a session starts, measure the time difference between PC and phone clocks (e.g. phone sends an echo message or timestamp on start command). Record this offset in the session's `sync_data.csv` for transparency [4]. Use it to adjust incoming Android data timestamps on the fly. Verifying this calibration ensures the Android data (GSR, PPG, etc.) aligns precisely with PC timestamps from the outset. (If the offset is significant, warn the user or automatically compensate in the data logger.)

3. **Enable Periodic Re-Synchronization (Drift Compensation):** For longer recordings, implement periodic sync pulses or timestamp exchanges to detect clock drift between devices. For example, send a `sync_time` command every few minutes and compute any deviation between PC time and phone time since the last sync. If drift is detected, apply a slight adjustment to incoming timestamps or at least log the drift in `sync_analysis.json` for later correction. This will keep multi-device timing tight over extended sessions (important if recording for tens of minutes or hours). The design can leverage Android's **SynchronizationManager** to handle network sync events in real-time [5] [1].

4. **Guarantee Synchronized Start/Stop Triggers:** Ensure the **fyp-gsr-windows** controller sends start/stop **commands** to the Android app(s) in a coordinated manner so all devices begin and end recording simultaneously [6]. Use the dual-phone manager's broadcast capability to send a unified `start_recording` action to both phones at the same moment [7]. On Android, leverage the CountDownLatch in its RecordingController so that all modules (thermal, RGB, GSR, audio) start capturing together once the start signal is received [8]. Similarly, coordinate the stop command so no device stops prematurely. This will minimize any start/stop timing skew across platforms (targeting millisecond-level difference at most).

5. **Harden Network Communication & Error Recovery:** Improve the reliability of the TCP/UDP link between PC and Android:

6. **Retry and Reconnect:** Implement automatic reconnection logic if a phone's connection drops mid-session. The system should attempt to reconnect (over Wi-Fi or fallback to Bluetooth) without user intervention [9]. Test this by forcibly disconnecting a phone and ensuring the PC server threads handle it gracefully (no crashes or hangs, and data logging resumes on reconnection).
7. **Packet Loss Handling:** Add sequence numbers or acknowledgements for critical data packets from the phone. If a packet is missed or out-of-order, detect it and request a resend or mark the data gap [10]. This is important for GSR/PPG streams at 128 Hz – even a small network hiccup could drop samples. The Android app's NetworkingManager can be extended to support ACK/NAK or to send redundant sync markers for verification.

8. **Latency Minimization:** Optimize the network stack to reduce delay. For Wi-Fi, prefer a persistent TCP socket for commands and possibly use UDP for streaming high-rate sensor data to lower overhead (with the above loss handling in place). Ensure the PC's Android receiver runs in its own thread or async event loop so that incoming data is processed promptly on arrival (preventing any 50+ ms queuing delays). The goal is to keep end-to-end latency as low as possible (the team observed ~50 ms latency over Wi-Fi, which is acceptable but could be improved) [11].

9. **Integrate Android Sensor Data into PC Logging Pipeline:** Merge the incoming Android data streams cleanly with the PC's data logging. Use the DataLogger to write phone-provided events to the `android_data.csv` with proper timestamps and labels [12]. Ensure that each data type from Android (e.g. GSR value, PPG value, phone status events) is captured:

10. Parse the JSON data events from Android (e.g. via the AndroidStreamReceiver). For each GSR sample or PPG reading coming in, log it with the synchronized timestamp (after applying any offset calibration from task 2).
11. If the Android app provides derived metrics like heart rate, incorporate those as well (possibly as a separate column or in the session metadata). This way the PC session folder contains all relevant phone data in a structured format for analysis.

12. Verify that the **data rates** match expectations – e.g. ~128 Hz for GSR from the phone [13]. If the phone's sampling produces slight timestamp jitter, consider buffering and smoothing the timestamps on the PC side (or use the phone's sequence indices to detect if any sample was delayed or lost).

13. **Synchronize Multi-Modal Streams in the PC Timeline:** In the capture controller, ensure that all data (local RGB, local thermal, local GSR, plus remote Android streams) are timestamped relative to the same clock. The PC's C++ backend already uses a central high-precision clock for local video and sensor capture [14]. Extend the synchronization module so that when Android data arrives, it gets aligned to that same clock (using the offset from task 2). This might involve stamping incoming packets with the PC receive time as well and comparing to the phone's embedded timestamp. Any consistent lag (e.g. if every phone sample is ~X ms behind) could be subtracted out to improve alignment. The end result should be that a given real-world event has matching timestamps in PC and Android data to within a few milliseconds. (For example, a GSR spike caused by a stimulus appears nearly concurrent in the PC's timeline and the phone's log.)

14. **Refine Threading and Concurrency for Dual-Device Capture:** Audit and improve how the system handles multiple input streams in parallel:

15. Make sure the network receiver for Android data is running on a **separate thread** (or one per phone) so that heavy network traffic from one phone doesn't block the GUI or local data capture. The DualPhoneManager should spawn independent client handler threads for Phone1 and Phone2 to process their data concurrently [15] [16].
16. Protect shared resources with locks where appropriate (e.g., if the DataLogger is writing to disk from both local and remote data sources). Use producer-consumer queues if needed: one thread can queue incoming phone data and the logger thread can batch-write to disk to avoid I/O contention.

17. Test with two phones streaming simultaneously to ensure the PC can ingest both without performance degradation. For example, if both phones send 128 Hz GSR and possibly other events, the PC should handle ~256+ events per second on top of video. Optimize the code path (e.g., minimal data copying, use efficient JSON parsing or a binary protocol if JSON proves too slow). The aim is to keep PC CPU overhead low so that adding phones doesn't risk local frame drops.

18. **Ensure Configurability and Conflict Prevention:** Update the configuration management so that using an Android device does not conflict with local sensors:

19. If `android_sync_enabled: true` (or dual_phone mode is on), the software should **optionally disable local GSR/PPG capture** to avoid fighting over the same sensor. For instance, if a Shimmer GSR device is being used via the phone's Bluetooth, the PC shouldn't also try to open a COM port for it. Implement logic in `src/config.py` or the CaptureController to detect this scenario and log a warning or automatically turn off PC's direct GSR capture when a phone is providing that data.
20. Expose clear settings in the UI's SettingsDialog for the user to indicate where each data source is coming from (PC or Android). For example, a checkbox for "Use Android for GSR/PPG" could switch the system to expect those streams from the network instead of local COM ports. Ensuring the config is mutually consistent will prevent runtime errors and make the setup more user-friendly.

21. Expand unit tests for configuration (like those in `validate_android_integration.py`) to cover these new cases, verifying that invalid combos are caught (e.g., dual_phone_enabled + GSR sensor port both set) [17] [18].

22. **Verify and Optimize Timestamp Alignment Logic:** After implementing the above, perform a thorough review of how timestamps are generated and transformed:

    - In the C++ capture engine, confirm that all frames and sensor samples get a timestamp from the same high-res clock (they achieved ~1 ms alignment between RGB and thermal on PC) [19]. No changes likely needed there, but ensure Python doesn't inadvertently re-timestamp frames when logging – it should use the provided timestamps from C++.
    - In the Android app, double-check that all data events use `elapsedRealtimeNanos()` (already indicated in the design) for consistency [20] [21]. This ensures phone data is internally synced. The main work is handling the offset on the PC side.
    - Optimize any math conversions: e.g., if converting Android nanosecond timestamps to PC seconds, use double precision to avoid truncation. Also consider the clock drift: if after a long recording the drift is known, you might apply a linear correction to timestamps in post-processing (this could be part of the **sync_analysis.json** generation).

- Finally, verify that `sync_analysis.json` includes useful metrics (e.g., start offset, end offset, drift, max deviation between streams) and extend it if not. This file can help quantify the sync quality achieved in each session.

Performance and Data Handling Enhancements

1. **Optimize High-Bandwidth Data Processing:** Ensure the system can handle the full data load without bottlenecks:

    - Profile disk I/O and CPU usage when recording 4K RGB + thermal video + GSR. The design already uses on-the-fly video encoding (H264) in C++ and asynchronous writing, which kept 10 min tests smooth with no frame drops [22]. Verify this still holds when Android data is also streaming in. The additional load from a 128 Hz sensor stream and occasional status packets is small, but test under maximum settings (e.g. 30 FPS video, 128 Hz GSR) to be safe.
    - If any performance issues arise (e.g., high CPU in Python), consider offloading work: for example, use Python's `asyncio` or multiprocessing for writing data to disk so the main thread isn't blocked. The PC research report notes the importance of keeping the Python layer "thin" so as not to disturb the C++ timing [23]. Maintain this principle by doing as much as possible in background threads (or in the C++ process) especially when handling incoming network data or heavy JSON serialization.
    - **Thread Priorities:** Investigate raising the priority of critical threads (if OS allows) – e.g., the C++ capture thread and the Android receiver thread – to reduce latency/jitter on Windows. This is a lower-level tweak, but on Windows one can use `SetPriorityClass` for the process or real-time priority for certain threads. This, combined with advising users to use the High Performance power plan, can further stabilize timing [24]. Document any such recommendations in the user guide.

2. **Improve Real-Time Sync Monitoring in the UI:** Extend the **Synchronization Visualizer** in the GUI to incorporate Android data streams [25]. Currently, the sync visualizer/timeline likely shows local RGB, thermal, GSR, PPG timelines and flags if they skew. Update it to also display markers for Android events:

    - For example, when the phone sends a periodic sync marker or its first data sample, plot that on the timeline against the PC's data. If any noticeable lag is present, the UI could highlight it (e.g., "Phone 1 data is 40 ms behind PC clock") so the user gets immediate feedback on synchronization status.
    - Show phone connection latency statistics in real-time (the DualPhoneManager keeps stats like last data time, bytes, etc.) [26]. Perhaps in the Dual Phones tab or status bar, display the current round-trip time or last sync delta. This would help during setup – if Wi-Fi is congested and latency spikes, the user can see it and adjust (or switch to wired).
    - Ensure that the GSR/PPG **plot widgets** can accept data from remote sources. If not already, add an option to plot "Phone GSR" vs "PC GSR". If the PC isn't using a local GSR device, the phone's GSR could just feed the existing plot as if it were local (perhaps differentiate it with a legend). This gives the operator live confirmation that the phone's physiological data is coming in and aligned (e.g., they'll see the GSR curve update in near-real-time, confirming sync).

3. **Enhance Sensor Data Handling and Quality:** Introduce improvements for sensor stream robustness and accuracy:

    - **Consistency Checks:** Implement sanity checks on sensor values and sampling interval. For instance, verify the GSR values from Shimmer (via phone) stay within expected ranges and flag if there's a dropout (no new sample for >1/128 sec) or spike (which might indicate a sync issue or sensor error). Log any anomalies in the session log or UI.
    - **Unit Standardization:** Confirm both PC and Android are using the same units for GSR (e.g., microsiemens) and PPG (e.g., raw or BPM for heart rate). The Android app already converts GSR to µS [13] ; ensure the PC does the same for its data or notes the units. For PPG, if the phone computes BPM in real-time [27] , consider logging both the raw PPG amplitude and the BPM. The PC's data format could be extended to include a heart rate column in `ppg.csv` or included in `android_data.csv`.
    - **Timebase Alignment for GSR Samples:** Since GSR is slower than video, the PC currently aligns GSR by interpolation to the video timeline [28] . With phone-provided GSR, do similarly – e.g., each GSR sample gets a precise timestamp; in analysis one can interpolate or find nearest video frame. No code change needed if timestamps are correct, but ensure documentation instructs how to align them. If desired, implement a simple post-process that marks which video frame index each GSR sample corresponds to (this could go into `sync_analysis.json` for convenience).
    - **Optional Smoothing/Filtering:** As a lower priority enhancement, consider offering basic filtering for physiological signals (e.g., a moving average for GSR or a peak detection verification for PPG) to improve data quality. This doesn't directly affect sync, but a stable signal can make manual sync verification easier (less noise when comparing events).

4. **Conduct Thorough Dual-Device Testing and Tuning:** Before considering the sync implementation "reliable", run extensive tests with different scenarios:

    - **Single Android Device:** Test with one phone acting as data source (GSR/PPG + cameras) and the PC capturing video. Simulate various network conditions (good Wi-Fi vs. limited bandwidth or even using Bluetooth only). Ensure that even over Bluetooth (which has higher latency and lower throughput), the system still functions (maybe at reduced video quality or slower updates). Optimize the code paths for Bluetooth if needed (e.g., smaller packets).
    - **Dual Android Devices:** Connect two phones simultaneously (as supported by the new DualPhoneManager) [29] . Verify each phone's data is correctly identified and logged (no mix-up between phone1 and phone2 data). Test sending independent commands (e.g., start one phone slightly later) and confirm the PC handles it. If any race conditions or thread contention appear with two connections, fix them (for example, ensure thread-safe access when both phones send data at the same time). The result should be that two phones can record in sync with the PC just as reliably as a single phone.
    - **Maxed-Out Use-Case:** If possible, test the absolute limits: e.g., PC recording two video streams + local GSR *and* two phones streaming their own sensors (and perhaps video, if ever implemented). While this may not be a normal use, it can reveal any hidden bottlenecks or memory leaks when the system is under stress. Monitor CPU, memory, and network usage during these tests and optimize accordingly (e.g., if memory usage grows, ensure buffers are cleared or reused).

Lower-Priority Feature Enhancements

1. **Implement Automatic Device Discovery and Simplified Pairing:** To improve usability, allow the PC to find and connect to Android devices without manual IP entry:

    - Use Bluetooth discovery or mDNS over Wi-Fi to detect phones running the app. For example, the Android app could advertise a service name on the network; the PC could scan and list available devices. This addresses the future enhancement of **phone discovery automation** [30] .
    - Provide a UI flow to pair a new phone: the user might click “Scan for devices”, select the phone (identified by name or ID), and then the system configures the IP/MAC in config.json automatically. This reduces setup errors and speeds up deployment in new environments.
    - If feasible, integrate a **QR code** or similar in the phone app that the PC’s webcam can scan to get connection info (an idea to make pairing even more seamless). This is an enhancement for convenience and does not affect core sync but improves the overall system integration.

2. **Expand Remote Command and Control Features:** Build on the existing command system to support more actions and configurations sent from PC to Android:

    - Implement **sensor configuration commands** on the PC side to remotely adjust phone settings (as hinted by the custom command example in the docs) [31] . For instance, a command to toggle phone camera modes (switch between front/back camera or RGB/thermal) or to change the GSR sampling rate from the PC UI would enhance integration.
    - Add a **time synchronization status query** command – the PC could ask the phone to report its current clock or send a test marker, which the PC then uses to evaluate latency on demand. This complements automated sync but can be a useful debugging tool.
    - Consider commands for starting calibration routines on the phone (e.g., ask the phone to flash its flashlight or display a pattern for calibration – see checkerboard sync idea below). This ties into experimental prep, giving the researcher a way to trigger sync cues on all devices simultaneously from the PC.

3. **Support Additional Devices and Modalities (Future-Proofing):** While currently supporting up to 2 phones, design the network module with scalability in mind:

    - Abstract the DualPhoneManager to a more general **MultiDeviceManager** that could potentially handle N devices in the future. This would involve using dynamic lists instead of fixed “phone1/phone2” slots. Although extending beyond 2 is low priority now, laying the groundwork (such as avoiding hard-coded limits in data structures) could ease future expansion [32] .
    - Extend support for other sensor types that an Android might provide. For example, if in the future an Android device also streams accelerometer or GPS data along with GSR, ensure the system can accommodate new data fields. This may simply require making the Android data handling generic (e.g., processing any JSON DataEvent the phone sends). The config could list expected phone data types to log. This makes the system more flexible for research needs down the line.
    - **Cloud/Remote Integration:** As a very low priority idea, explore connecting the system over the internet (cloud server or VPN) so that an Android in the field could send data to a lab PC remotely. This would involve security (authentication, encryption) and dealing with higher latencies, so it’s a significant project. Nonetheless, structuring the network code to

allow a configurable server address (not just localhost or LAN) could be a first step toward this “cloud integration” vision.

4. **Optional Hardware Sync Mechanisms:** Although the system achieved good sync via software, consider providing hooks for external synchronization if needed by advanced users [33]. For example:

    - Allow the PC to output a trigger signal (e.g., via an Arduino or parallel port) that can be used to **genlock cameras or mark sensors**. The FLIR thermal camera supports an external trigger input [34]. Implement a module that toggles a digital line at start, which could simultaneously flash an LED or send a pulse to any device that can accept it (some GSR devices can take event markers [35]). This task would involve hardware and is optional, but documenting how one could use it with our system adds value for users who require microsecond-level sync.
    - Similarly, if multiple PCs were ever used (e.g., one PC per camera), an output trigger or an LSL synchronization pulse could coordinate them. This is beyond the current scope but worth noting as a future extension for ultra-high precision setups.

Experimental Setup and Validation Tasks

(These final tasks focus on research preparation and validation to ensure the synchronized system works as expected in practice.)

1. **Frame Synchronization Testing with Visual Markers:** Conduct controlled tests using an **LED flash or checkerboard pattern** visible to all cameras (PC’s RGB, PC’s thermal, and the Android’s cameras) to verify frame-level sync:

    - Set up an LED that is triggered to blink at a known time (e.g., the moment recording starts or at a specific timestamp). Record a short session where this LED flash occurs in view of all cameras. After recording, examine the video frames to see if the flash appears at the same frame (or within one frame) in each video stream. The expectation is that the difference should be on the order of one frame or less (around 33 ms at 30 FPS, ideally much lower) [36]. In earlier tests, the flash was captured within the same frame index or at most one frame apart between RGB and thermal, with timestamp log differences ~5–10 ms [36] – use this as a benchmark.
    - Repeat this test for both the PC and phone. If the Android phone’s RGB or thermal camera also sees the flash, compare the phone’s recorded frame timestamp to the PC’s. This will directly validate the cross-device sync. Ideally, the phone’s flash frame timestamp (after offset adjustment) should match the PC’s flash frame timestamp within a few tens of milliseconds (taking into account network delay). Any larger discrepancy indicates a need to adjust the sync offset or investigate timing logs.
    - **Checkerboard pattern**: Alternatively or additionally, display a high-contrast checkerboard on a tablet or screen that is in view of all cameras and have it **change suddenly** (e.g., invert from black to white) to serve as a sync signal. This can be easier for thermal cameras to pick up if the screen emits IR differently on a bright image. Use high-speed video if available to pinpoint the timing. This approach, like the LED, provides a ground truth visual marker across devices.

2. **Verify Timestamp Accuracy and Multi-Stream Alignment:** Design procedures to confirm that sensor data and video data are correctly aligned in time:

    - Use a known physiological event alongside a visual cue. For example, have a participant (or tester) quickly **tap the GSR electrodes** or perform a quick deep breath to create a spike in the GSR signal, while simultaneously triggering a visual marker (like an LED) [37]. In the recorded data, check that the GSR spike's timestamp corresponds to the moment of the LED flash in the video frames. Previous tests showed GSR spikes occurred within a 1–2 frame window of a simultaneously observed flash, with any constant offset ~10 ms (due to sensor processing) [38]. Ensure your system still meets this: if not, determine whether the offset is larger and whether it's consistent (which could be corrected in software).
    - If the Android is providing GSR, do the same with the phone's GSR data vs. the PC's video. This will validate cross-platform alignment. Compare the phone's logged timestamp for the spike event to the PC video frame time of the flash. If you find, say, a 50 ms delay (as was observed for phone-streamed events over Wi-Fi) [11], confirm that this matches the network latency and is accounted for in your synchronization (the phone data might consistently lag by 50 ms – if so, you might decide to subtract a fixed 50 ms in the timestamps to compensate).
    - **Audio synchronization (if applicable)**: If the Android app records audio and a sharp sound (e.g., clap) is made during the test, compare the audio waveform's timestamp to the video flash. The phone logs audio, but the PC doesn't currently – however, if needed, you could record audio on the PC as well for one test (even using a simple microphone and recording utility) to have another comparison of timing. This is optional, but audio can be a convenient sync reference since it's easy to match a clap sound across devices.

3. **Conduct Mock Experimental Runs for Stress Testing:** Before deploying the system in real research, do full-length trial runs:

    - Simulate a typical experiment session (for instance, record for 30 minutes with all modalities active) to see if any issues arise over time. Monitor the system for **frame drops or slowdowns**. The system should ideally log zero frame drops (internal counters should show no gaps) and stable sampling rates over such duration [39]. If you plan longer recordings (1+ hour), test those as well to observe if any drift accumulates or memory usage increases. Because the team didn't extensively test multi-hour sessions [22], it's important to do so now and be confident in stability. If you discover a slow memory leak or increasing sync error, address it before real experiments.
    - Perform a stress test under heavy load conditions: for example, start recording with two phones connected, then open other programs or simulate background CPU load on the PC (though in actual use you'd avoid this, it's good to see how much headroom exists). The system was around 50–70% CPU in tests with all features on [33]; verify similar or better performance on your setup. Ensure Windows doesn't throttle the process – use **Task Manager** or performance monitors during the run.
    - Test failure scenarios: e.g., disconnect a sensor or turn off a phone in the middle of recording (a "pull the plug" test). The system should handle it gracefully (e.g., log an error but continue running). If the phone reconnects, see if the session can continue. These drills will expose any error-handling bugs under real conditions. Make improvements as needed (for instance, if stopping the recording hangs because it's waiting on a dead connection, fix the shutdown logic to time out properly).

4. **Refine Experimental Protocols for Sync Validation:** Develop a standard procedure that researchers can use each time to validate sync just before data collection:

    - For instance, at the start of each recording day, do a quick **LED flash test** (as in task 19) to ensure all devices are syncing properly. This can be as simple as a smartphone flash or a small LED device that you trigger and then verify the PC and phone times match within the expected range (you might even build a tiny utility that automatically analyzes the flash frame timestamps across video files).
    - Create a **checklist** (to include in documentation) that covers syncing steps: e.g., "Ensure PC and phones are on the same NTP time server (if using system clock), ensure Wi-Fi signal is strong, do a test recording with a known marker, check that `sync_analysis.json` reports sync error < X ms, etc." This isn't a code task per se, but it's an important part of research preparation to guarantee data quality.
    - Train the team on interpreting the sync metrics. For example, if `sync_analysis.json` yields a max deviation of 20 ms and drift of 5 ms over 30 min, confirm that this is acceptable for the experimental goals. If a stricter sync is needed, decide on action (maybe use a wired sync trigger or ensure all devices run on the same hardware clock via external trigger).
    - Document any limitations discovered. If, for instance, you find that Bluetooth-only connections introduce too much latency or occasional sample loss, note that in the experimental plan (maybe you'll mandate Wi-Fi for critical measurements). The goal is to leave no surprises when the actual data collection is underway.

5. **Prepare and Test Synchronization Aids (if needed):** Based on the above validation, decide if any extra synchronization aid is necessary and experiment with it:

    - If timing differences are slightly higher than desired, one simple aid is to use a **visual metronome or flashing screen** during recording. For example, a tablet that flashes a colored square every 5 seconds placed in view can later help align streams during analysis (by comparing the timing of flashes in each video). This can serve as a backup in case software timestamps need verification.
    - Another approach is a **physical sync device**: for example, an Arduino that drives an LED and simultaneously sends a signal to a PC input (or even an LSL marker) could provide a precise reference. You might build a prototype where pressing a button triggers: a) an LED flash, b) a TTL pulse to an input that the PC can timestamp (if you have a DAQ or even using the audio input), and c) perhaps a Bluetooth signal that the phone app can detect. This is complex and not typically required given the <50 ms software sync achieved, but testing it in a lab environment could be a worthwhile experiment to see if it further reduces uncertainty [35].
    - Evaluate the outcomes of these experiments. If the LED flash method already shows <10 ms differences, additional hardware might not improve much and can be kept in reserve. On the other hand, if a particular study demands absolute certainty of sync, having tested options for hardware triggering gives you a plan to meet those requirements.

By addressing the tasks above in order of priority, **fyp-gsr-windows** will be more robust and tightly integrated with **fyp-gsr-android**. High-priority fixes will ensure that all data streams (RGB video, thermal video, GSR, PPG, etc.) are **synchronized to a common timeline** with high precision across the PC and Android devices. Lower-priority enhancements will improve ease of use, performance overhead, and extend the system's capabilities (e.g., scalability and advanced features), while the experimental setup tasks will verify that the whole system performs reliably in practice. Following this plan will result in a dependable multimodal data acquisition setup ready for rigorous research data collection.

Sources:

- PC System README – *GSR-RGBT Windows Data Acquisition System* (features & architecture) [40] [14]
- Android App README – *GSR-RGBT Android Data Capture (synchronization capabilities)* [13] [8]
- PC Research Report – *Timing accuracy, sync tests (LED flash, GSR alignment), performance stats* [36] [41]
- Dual Phone Implementation Docs – *Network sync commands and error handling* [6] [9]

---

[1] [11] [14] [19] [22] [23] [24] [28] [33] [34] [35] [36] [37] [38] [39] [41] pc_research.tex
https://github.com/buccancs/fyp-gsr-windows/blob/4cd1d1074afcabad6987ac5bec39dcc8df22a644/docs/pc_research.tex

[2] [6] [7] [9] [10] [15] [16] [26] [30] [31] [32] DUAL_PHONE_IMPLEMENTATION.md
https://github.com/buccancs/fyp-gsr-windows/blob/9942a9bc58cffdc2b9badc564115c116f81bd035/DUAL_PHONE_IMPLEMENTATION.md

[3] [4] [12] [25] [40] README.md
https://github.com/buccancs/fyp-gsr-windows/blob/9942a9bc58cffdc2b9badc564115c116f81bd035/README.md

[5] [8] [13] [20] [27] README.md
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/README.md

[17] [18] validate_android_integration.py
https://github.com/buccancs/fyp-gsr-windows/blob/9942a9bc58cffdc2b9badc564115c116f81bd035/tests/validation/validate_android_integration.py

[21] SynchronizationManager.kt
https://github.com/buccancs/fyp-gsr-android/blob/bd9828a3a2899254d1e2a1e042cdd70175637d52/app/src/main/java/com/gsrrgbt/android/sync/SynchronizationManager.kt

[29] CONTINUATION_SUMMARY.md
https://github.com/buccancs/fyp-gsr-windows/blob/9942a9bc58cffdc2b9badc564115c116f81bd035/CONTINUATION_SUMMARY.md