# Implementation Guide: Shimmer3 GSR+ Support (Milestone 2.4)

This guide outlines the design and implementation for **Milestone 2.4** of the Android synchronized recording app, focusing on integration of **Shimmer3 GSR+** wearable sensors. The goal is to support multiple Shimmer3 GSR+ devices streaming concurrently alongside existing RGB and thermal recorders, with synchronized data capture. We will cover the ShimmerRecorder component design, Bluetooth pairing workflow, multi-device management, channel configuration, data logging, PC streaming, timestamping, threading model, SessionInfo integration, reliability, and a manual test plan.

## 1. ShimmerRecorder Class Structure and API

**Class Role:** ShimmerRecorder will encapsulate all functionality for discovering, connecting, and recording data from multiple Shimmer3 GSR+ sensors. It acts as a manager for Shimmer devices, similar to existing recorders (RGB camera, thermal camera), enabling concurrent operation. This class ensures modularity by isolating Shimmer-specific logic and provides a clean public API to start/stop recording and access data.

**Key Responsibilities:**

* Maintain a **list of connected Shimmer devices** (or a map from device ID to Shimmer instance).
* Provide methods to **discover and pair devices** via Bluetooth.
* Allow enabling/disabling specific sensor channels per device before streaming.
* Start and stop streaming for all connected Shimmers in sync with the session lifecycle.
* Log incoming sensor data to files and forward data to the PC in real time.
* Handle device state changes (e.g., connection, disconnection, errors) and attempt reconnection.
* Clean up connections and file handles on session end.

**Public API (Methods):**

* scanAndPairDevices(): Launches the device discovery UI or workflow to find Shimmer devices and pair if needed.
* connectDevices(List<String> deviceAddresses): Connects to multiple Shimmer3 devices by their MAC addresses. Internally, it will create Shimmer instances for each and establish Bluetooth connections.
* setEnabledChannels(String deviceId, Set<SensorChannel> channels): Configures which sensor channels are active on a given device (e.g., GSR, PPG, accelerometer). This will construct the appropriate sensor bitmask and send configuration to the device.
* startRecording(SessionInfo session): Begins streaming from all connected Shimmers, starts data logging (using paths from SessionInfo), and starts forwarding data to PC. This should coordinate with RGB/Thermal recorders so that all start simultaneously.
* stopRecording(): Stops streaming on all devices, closes log files, and stops any network streams. Prepares the system for a new session or app exit.
* disconnectAll(): Gracefully disconnects all Shimmer devices (could be called on session end or app shutdown).
* **Callback/Listeners:** The class can use a listener or broadcast system to notify of important events (e.g., device connected, data received, device disconnected) if other components or UI need updates (for example, to display live values or alert the user of a disconnect).

**Internals and Structure:**

* It will likely use the Shimmer Android API’s Bluetooth manager for multi-device support. For example, the Shimmer API provides a ShimmerBluetoothManagerAndroid to manage multiple units[[1]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerConnectionTest/src/main/java/shimmerresearch/com/shimmerconnectiontest/MainActivity.java#L360-L365). ShimmerRecorder can hold an instance of this manager to track devices. Each Shimmer device will be represented by a Shimmer object (from the Shimmer SDK) that handles communication with that sensor.
* A single Android Handler (or Kotlin coroutine channel) will receive data callbacks from all Shimmer devices. The Shimmer API is designed to send asynchronous messages for sensor data and state changes[[2]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L150-L158). We will use this to route data to log files and network, and to handle events like disconnects.
* The class should be implemented as a singleton or as part of a higher-level RecordingSession controller so that other components (SessionInfo, UI, etc.) can interact with it.

By designing ShimmerRecorder as a standalone module with a clear API, we ensure the solution is **modular and scalable**. New sensor types or additional Shimmer devices can be integrated with minimal changes, and the class can be tested in isolation.

*(The Shimmer Android API supports full configuration of sensors and simultaneous capture from multiple Shimmer units, which we leverage in this design*[*[3]*](https://www.shimmersensing.com/product/shimmer-java-android-api/#:~:text=,Auto%20calibration%20of%20data)*.)*

## 2. Bluetooth Permissions and Pairing Workflow

Support for Shimmer3 sensors requires managing Bluetooth Classic connections. We must handle Bluetooth permissions and device pairing in the Android app:

* **Runtime Permissions:** On Android 12 (API 31) and above, the app needs the BLUETOOTH\_SCAN and BLUETOOTH\_CONNECT permissions to search for and connect to devices. Additionally, location permissions (ACCESS\_FINE\_LOCATION and ACCESS\_COARSE\_LOCATION) are required for Bluetooth device discovery (scanning) on Android 6.0+ and remain needed even on newer versions for BLE or discoverability. The app should check for these at startup and request them if not granted[[4]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L53-L61)[[5]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L79-L87). The code will look for BLUETOOTH\_CONNECT/SCAN on Android S+, and for older versions use location perms, as shown in the ShimmerBasicExample code[[4]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L53-L61). If not already granted, request the user’s permission (present a system dialog).
* **Enabling Bluetooth:** If the device’s Bluetooth is off, prompt the user to enable it. This can be done via an ACTION\_REQUEST\_ENABLE intent to open Bluetooth settings. This step is typically handled prior to scanning.
* **Device Pairing:** Shimmer3 devices must be paired with the Android host. The Shimmer3 GSR+ uses a default PIN code for pairing. When the app scans and discovers a Shimmer that isn’t paired, it should initiate pairing. The user will be prompted to enter the PIN – by Shimmer default, **the PIN is 1234**[[6]](https://shimmersensing.com/wp-content/docs/support/getting-started/Streaming_to_Android.pdf#:~:text=The%20PIN%20is%201234,PAIRING%20THE%20SHIMMER%20OVER%20BLUETOOTH). Once entered, the Android system will bond with the device. (Alternatively, instruct users to pre-pair devices in Bluetooth settings using the same PIN.)
* **Discovery UI:** Provide a UI to list available Shimmer devices. We can utilize the Shimmer API’s built-in discovery dialog ShimmerBluetoothDialog for convenience. Calling startActivityForResult with this dialog will show a list of paired devices and an option to scan for new devices[[7]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L127-L134). The user selects a device from the list. For multi-device support, we may allow repeated use of this dialog or a custom multi-select list:
* For example, have a “Add Shimmer Device” button that opens the scanner. The user picks a device; the app stores the returned MAC address. The user can repeat to add another device.
* All selected devices can then be connected for streaming.
* **Handling onActivityResult:** After the user chooses a device in the dialog, the result callback provides the device MAC address[[8]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L250-L258). The app should instantiate a new Shimmer object (or use the manager) and call connect on that address. We will do this for each device chosen. It’s important to handle the result asynchronously (onActivityResult or equivalent callback in Jetpack’s Activity Result API) because scanning is asynchronous.
* **Bonding Workflow:** In some cases, if the device was not paired, the ShimmerBluetoothDialog’s scan will trigger the pairing flow (with the PIN entry). We must ensure the app has BLUETOOTH\_ADMIN permission if using older APIs for pairing (on newer Android, pairing is covered by BLUETOOTH\_CONNECT). Once paired, we can proceed to connect.

In summary, the app must **request required Bluetooth permissions** at runtime, ensure Bluetooth is enabled, then **discover and pair with Shimmer3 devices**. This can be done via the provided Shimmer scan dialog or a custom scanner that finds devices advertising the Shimmer service. After this, we obtain the MAC addresses of target devices for the next step.

## 3. Device Discovery and Multi-Shimmer Connection Logic

With permissions granted and devices paired, the ShimmerRecorder will handle connecting to multiple Shimmers concurrently:

* **Device Identification:** Shimmer3 GSR+ devices can be identified by their Bluetooth names or MAC addresses. Often, they have default names like “Shimmer” or a device ID. The user should select the specific devices to use for the session (especially if multiple Shimmers are in range).
* **Connecting Multiple Devices:** We utilize the Shimmer API to manage multiple connections. The Shimmer SDK allows simultaneous Bluetooth streams from multiple units[[9]](https://www.shimmersensing.com/product/shimmer-java-android-api/#:~:text=,Auto%20calibration%20of%20data). Internally, we will create a Shimmer instance for each device. If using ShimmerBluetoothManagerAndroid, we register each Shimmer device with the manager and connect via its MAC:
* For each selected MAC, do something like: Shimmer shimmer = new Shimmer(handler, context); shimmer.setMacIdFromUart(mac); btManager.putShimmerGlobalMap(mac, shimmer); btManager.connectShimmerThroughBTAddress(mac);[[1]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerConnectionTest/src/main/java/shimmerresearch/com/shimmerconnectiontest/MainActivity.java#L360-L365). This sequence (from Shimmer’s connection test) demonstrates adding a Shimmer device to the manager’s map and initiating connection.
* The manager helps keep track of all active connections in a central place (a global map of MAC -> Shimmer device).
* **Connection Sequence:** It’s wise to connect devices one by one (to avoid overwhelming Bluetooth). The connectDevices() method of ShimmerRecorder can iterate through the MAC list and call connect for each. The Shimmer API’s callbacks will inform when each is connected. Once connected (and fully initialized), we can then start streaming. We should wait until all desired devices are connected before commencing the unified recording session.
* **State Management:** The ShimmerRecorder should maintain a list or map of active devices. Each entry can contain:
* Device MAC or ID,
* A reference to the Shimmer object or a wrapper (let’s call it ShimmerDevice if we wrap extra info),
* Current status (connected, streaming, etc.),
* What sensors are enabled on it.
* **Failure Handling:** If a device fails to connect (e.g., out of range or wrong PIN), the app should report it to the user and possibly allow retry. The connection process might time out or throw an exception (Shimmer API throws ShimmerException on issues). Catch these and handle gracefully (skip that device or retry).
* **Multiple Connections in UI:** Ensure the UI reflects multiple devices. For example, show a list of connected Shimmer devices with their names or IDs, and maybe an indicator (LED icon) for connection state. This helps the user verify that, say, two devices (“Shimmer A” and “Shimmer B”) are ready.

By structuring the connection logic to handle a **dynamic list of devices**, we ensure scalability. Whether the user wants 1 or 5 Shimmer GSR+ units, the same flow can accommodate it. The Shimmer API is designed for multi-device streaming, as noted in their documentation (multiple Shimmer units can stream simultaneously to Android)[[9]](https://www.shimmersensing.com/product/shimmer-java-android-api/#:~:text=,Auto%20calibration%20of%20data).

*(The ShimmerBluetoothManagerAndroid class is utilized to manage multiple device connections concurrently*[*[1]*](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerConnectionTest/src/main/java/shimmerresearch/com/shimmerconnectiontest/MainActivity.java#L360-L365)*.)*

## 4. Channel Selection Toggles and Shimmer Configuration

Each Shimmer3 GSR+ sensor includes multiple sensing channels (e.g., **GSR** for skin conductance, **PPG** for photoplethysmograph/heart-rate, **accelerometer** axes for motion, etc.). It’s crucial to allow enabling or disabling specific channels per device to limit data to what’s needed and conserve bandwidth/battery.

**Available Channels on Shimmer3 GSR+:** Based on Shimmer3 specs, key sensor channels likely include: Accelerometer (3-axis), Gyroscope, Magnetometer, GSR (Galvanic Skin Response), and an optical pulse sensor (PPG/Heart Rate), among others. The Shimmer API defines bitmask constants for each sensor signal. For example, Shimmer.SENSOR\_GSR (0x04) controls the GSR channel, and there is a constant for a “Heart” sensor (PPG). Some relevant sensor codes from the API:

* Accelerometer = 0x80
* Gyroscope = 0x40
* Magnetometer = 0x20
* ECG = 0x10 (not used on GSR+ if no ECG module attached)
* EMG = 0x08 (also not on GSR+ unit)
* **GSR = 0x04**
* *Exp* channels 0x02, 0x01 for external ADC inputs (if used)
* Strain Gauge = 0x8000 (if applicable)
* **Heart Rate (PPG) = 0x4000** (often labeled as “Heart” in API)

Using these constants, we can build a bitmask representing all sensors to enable on a given device. For instance, to enable GSR and PPG on a Shimmer, the mask would be 0x04 | 0x4000. The Shimmer API allows setting this either in the connection call or via a method.

**Configuration Process:**

* Before streaming, for each device we determine which channels to record. The app UI can present toggles (switches or checkboxes) for each sensor type supported by that device. For Shimmer3 GSR+, likely toggle options: **GSR**, **PPG**, **Accel**, (and possibly **Gyro**, **Mag** if the unit has those; some Shimmer units have 9-DOF sensors included).
* Once the user selects the desired sensors, the ShimmerRecorder will configure the device:
* If using the Shimmer object directly: call something like shimmer.writeEnabledSensors(mask) or use the constructor that sets sensors. The Shimmer constructor has an overload that accepts setEnabledSensors bitmask, sampling rate, accel range, GSR range, etc[[10]](https://github.com/amaltesh/shimmer/blob/6181b187c489adec57fd0cb293b176da6aa4569a/ShimmerAndroidInstrumentDriver/src/com/shimmerresearch/android/Shimmer.java#L2-L10)[[11]](https://github.com/amaltesh/shimmer/blob/6181b187c489adec57fd0cb293b176da6aa4569a/ShimmerAndroidInstrumentDriver/src/com/shimmerresearch/android/Shimmer.java#L26-L34). We can use this to initialize the device with the proper sensor set. For example, new Shimmer(context, handler, "DeviceA", samplingRate, accelRange, gsrRange, enabledSensorsMask, false) – this would configure which channels are on.
* Alternatively, if we first connect with a default config, we can then call API methods to enable/disable sensors on the fly. The Shimmer API differentiates between “set” (in code) and “write” (apply to hardware) for configurations[[12]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/wiki#:~:text=In%20the%20Android%20API%2C%20you,to%20perform%20the%20same%20function)[[13]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/wiki#:~:text=The%20differences%20between%20these%20methods,are%20as%20follows). We should ensure to send the config to the device by using the proper write method or by including it at connect time.
* **GSR Range:** The GSR sensor typically has multiple excitation resistance settings (to accommodate different skin resistance ranges). The API likely has a parameter for GSR range (for example, 4.7 kΩ, 10 kΩ, etc). We should expose a selection for GSR range if needed (or choose a default like 4.7 kΩ which is common for skin conductance). In the Shimmer constructor call, there is a gsrRange parameter[[10]](https://github.com/amaltesh/shimmer/blob/6181b187c489adec57fd0cb293b176da6aa4569a/ShimmerAndroidInstrumentDriver/src/com/shimmerresearch/android/Shimmer.java#L2-L10) – for GSR+ we might set this to an appropriate constant (e.g., Configuration.GSR\_RANGE\_AUTO or a specific range).
* **Sampling Rate:** Ensure all Shimmers use the same sampling rate for synchronized capture. The API allows setting a sample rate in Hz. We might choose, say, 51.2 Hz or 128 Hz depending on use case. This can also be a user-selectable setting. The key is to use a uniform rate across devices and modalities (the cameras have their own frame rates, but for sensor data, consistency helps).
* **Accel Range:** Shimmer3 typically lets you choose accelerometer range (e.g., ±2g, ±4g, etc). A default (±2g) can be used or configured as needed.
* After configuring channels, when we call shimmer.startStreaming(), the device will only stream the selected sensors.

By toggling off unused channels, we reduce data load and focus on relevant signals. The ShimmerRecorder should store the chosen configuration (maybe in SessionInfo or internally) so that it knows what columns to expect in the data and how to log them.

*(The Shimmer Java/Android API provides full control over sensor configuration – e.g. enabling/disabling specific sensors like GSR or PPG*[*[3]*](https://www.shimmersensing.com/product/shimmer-java-android-api/#:~:text=,Auto%20calibration%20of%20data) *– using defined sensor bitmask constants.)*

## 5. Concurrent Data Logging and PC Streaming Pipeline

Once devices are streaming data, our system must handle two parallel tasks for each incoming data sample: **logging to local storage** and **streaming to the PC**, all in real time. The design must ensure these operations run concurrently without data loss.

**Data Reception:** The Shimmer API will deliver sensor readings asynchronously via the Handler callback. Each message (usually an ObjectCluster in Shimmer API terms) contains one sample’s data from one Shimmer device[[14]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L166-L174). As data arrives (which could be as fast as e.g. 50–128 samples per second per sensor), we must quickly process it.

**Logging to Local Storage:**

* For each Shimmer device, open a log file (e.g., a CSV file) in the session directory (obtained from SessionInfo, see section 8). Each file could be named after the device (e.g., Shimmer\_<DeviceName>\_<timestamp>.csv).
* Write a header line with column names (Timestamp, GSR, PPG, AccelX, AccelY, AccelZ, etc., depending on enabled sensors).
* On receiving a sample, format it as a new line in CSV:
* Use the session-relative timestamp or absolute time (see section 6 on timestamping).
* Include sensor values. The ObjectCluster provides calibrated sensor values by name[[15]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L170-L178). For example, objectCluster.getData(Configuration.Shimmer3.ObjectClusterSensorName.GSR) could retrieve the GSR value.
* Separate by commas and newline.
* Use a buffered writer for efficiency and flush periodically (to avoid data loss if app crashes mid-session).
* Ensure thread-safety: Since multiple device data could come in concurrently (in the Handler or separate threads), use synchronization or sequential processing for file writing. A simple approach is to handle all file writes in the single Handler thread (so they are serialized in the order data arrives). Alternatively, dedicate a background thread for file I/O and send it write tasks via a thread-safe queue.

**Streaming to PC in Real Time:**

* **Method:** Use a network connection to send data to a PC application live. The two main options are Bluetooth SPP or Wi-Fi (TCP/UDP). Given the phone is already using Bluetooth for Shimmer connections, using Wi-Fi (or USB tethering) is often more reliable for streaming out data.
* **Wi-Fi Socket:** The app can act as a client and send data to a known PC IP address over TCP or UDP. For example, open a TCP socket to the PC’s IP and port (assuming the PC is running a server or listening program)[[16]](https://dev.to/techkoool/how-to-transmit-android-real-time-sensor-data-to-computer-5308#:~:text=private%20class%20WifiAsyncTask%20extends%20AsyncTask,Socket%20socket%3B%20private%20PrintWriter%20printWriter). Once connected, send data lines as they come. This could simply mirror the CSV lines or use a lightweight protocol (e.g., JSON or CSV over socket).
* **Alternatively**, the app could open a server socket and let the PC connect as a client. However, firewall issues often make phone-as-client easier.
* **Bluetooth to PC:** If Wi-Fi is not available and the PC has Bluetooth, the app could connect to the PC over a Bluetooth socket (SPP profile). This requires pairing the phone with the PC and knowing the UUID of a custom service. It’s more complex and limited (and cannot use BLE). Wi-Fi is therefore recommended for high-throughput streaming.
* **Implementation:** Whichever method, use a **background thread** for network I/O:
* The ShimmerRecorder can start a NetworkStreamThread when recording begins. This thread establishes the socket connection (e.g., TCP to PC). Then it listens for data (perhaps via a queue or callback).
* On each sample, format the data similarly as for logging, and send it over the socket. For example, using a PrintWriter on the socket’s output stream to send a line of text[[17]](https://dev.to/techkoool/how-to-transmit-android-real-time-sensor-data-to-computer-5308#:~:text=1,connection%20using%20the%20PrintWriter%20object).
* Manage network errors: if the socket disconnects, attempt to reconnect or log an error. The user should be alerted if PC streaming fails, but local logging can still continue.
* **Bandwidth:** If multiple Shimmers at high sample rates and many channels are active, consider the network bandwidth. For example, 2 Shimmers \* 6 channels \* 50 Hz ~ 600 values per second – which is quite manageable (~tens of KB/s). Wi-Fi can easily handle this; Bluetooth might be tighter but still feasible if using a binary format.
* **Optional Data Aggregation:** If needed, the app could merge data from multiple devices into a single stream before sending to PC (to simplify PC-side). But merging would require aligning timestamps and buffering. A simpler approach is to stream each device’s data as a separate channel (e.g., tag each line with device ID and send interleaved).

In summary, the pipeline is: **Shimmer Device → Bluetooth → App (ShimmerRecorder) → [Log to File] and [Send to PC Socket]**. This happens for each sample in near-real-time. By using multithreading (Handler for intake, I/O thread for network, etc.), we ensure logging and streaming occur concurrently without slowing down data acquisition.

*(One approach is to use a socket connection over Wi-Fi and send sensor readings in real time as they arrive*[*[17]*](https://dev.to/techkoool/how-to-transmit-android-real-time-sensor-data-to-computer-5308#:~:text=1,connection%20using%20the%20PrintWriter%20object)*, while simultaneously writing them to local storage.)*

## 6. Timestamping Strategy and Session File Format

Accurate timestamping is vital for synchronized multi-modal recording. We need a strategy to timestamp Shimmer sensor data so that it aligns with other data (like video frames from RGB and thermal cameras) and across multiple Shimmer devices.

**Timestamp Sources:**

* **Device Timestamps:** The Shimmer3 firmware generates timestamps for each sample, often relative to its own start of streaming. The Shimmer API’s ObjectCluster provides a timestamp for each data packet (often labeled “Time Stamp” in the data cluster)[[14]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L166-L174). This is typically the device’s internal tick count (in milliseconds or an incrementing counter). For example, Shimmer might timestamp the sample at 1234.56 milliseconds since it started.
* **System Timestamps:** We can also capture the Android system time when data is received. This can be in UNIX epoch milliseconds (e.g., via System.currentTimeMillis() or a monotonic clock). System time allows correlating events across devices and with video frame timestamps (assuming the device clock is used for video timing).

**Chosen Approach:**

To maximize synchronization accuracy, it’s recommended to use the **Shimmer device timestamps** for internal consistency (they are recorded when the sample was taken on the sensor hardware, which avoids Bluetooth latency issues) and then map those onto a common timeline. Concretely:

* When a session starts, record a reference time (session start time) in epoch milliseconds.
* As each Shimmer starts streaming, note its initial device timestamp (which might be ~0 or some small value if the device resets at stream start).
* In each logged entry, include:
* The device timestamp (relative time from device’s perspective).
* A session timestamp (possibly calculated as sessionStartTime + deviceTimestamp if the device’s clock was aligned to start, or use an offset if there’s a known delay).
* Alternatively, include the system receive time as a separate column.
* For simplicity, we could log **only one timestamp column** that represents Unix time (epoch ms) of the sample. To derive that, add the device’s timestamp to the synchronized start time. However, any drift in the device clock vs. phone clock could introduce minor error. If high precision sync is required, it may be worth also logging the raw device timestamp.

**Session File Format:**

* Use a **CSV (Comma-Separated Values)** format for easy import to analysis tools. Each Shimmer device gets its own CSV file.
* Filename convention: include session identifier and device identifier. For example: Session123\_ShimmerDeviceA.csv.
* Header row: include column names. For example: Time(ms), GSR, PPG, AccelX, AccelY, AccelZ etc. The Time(ms) can be the unified timestamp in milliseconds since epoch (or since session start).
* Data rows: each row per sample with timestamp and sensor readings. Example:
* 1690488023456, 4.21, 76.3, 0.01, -0.03, 0.98  
  1690488023476, 4.18, 75.9, 0.02, -0.01, 0.99
* Here 1690488023456 would be a timestamp in Unix ms (which can be correlated to video timestamps).
* If separate timestamp columns are used (e.g., DeviceTime and SystemTime), document this in the header.

**Synchronization Considerations:**

* Start all Shimmer streams as close together as possible (ideally within the same few milliseconds). This way, device clocks start nearly in sync. If using the Shimmer manager, you could call btManager.connectShimmerThroughBTAddress for all, then once all are connected, quickly call startStreaming() on each back-to-back.
* Since the cameras likely use system time for frame timestamps, aligning sensor data to system time (via epoch timestamp) will make it easier to merge datasets.
* All logs in a session should use the **same time base**. Using epoch time in milliseconds is a good universal choice (e.g., an RGB frame at 1690488023500 ms vs a GSR sample at 1690488023456 ms can be lined up).
* Include timezone or time reference info if needed, but since analysis is relative, just ms ticks is fine.

**Example:** The ShimmerBasicExample prints the timestamp for each sample to logcat[[14]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L166-L174). We will instead write it to file. That timestamp is in the objectCluster as “CAL” (calibrated) time in milliseconds. We can retrieve it and transform as needed.

By adopting a clear timestamp strategy and documenting it in the file headers (e.g., “Time (ms since Unix epoch)”), we ensure data from Shimmers can be accurately synchronized with video and other modalities.

*(Each Shimmer data packet comes with a timestamp*[*[14]*](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L166-L174)*. We will convert those to a unified session timeline (milliseconds) for logging.)*

## 7. Coroutine/Threading Model for Managing Each Device

Managing multiple sensor streams and I/O in parallel calls for a robust threading (or coroutine) model. The design should avoid blocking the main UI thread and ensure each device’s data is handled promptly.

**Receiving Data (Threads vs. Handler):** The Shimmer API uses a callback/Handler mechanism to deliver data asynchronously. Under the hood, each Shimmer device spawns a worker thread to read from the Bluetooth socket and then posts messages to the provided Handler. In our design: - We can use **one Handler** for all Shimmer devices. For example, pass the same Handler mHandler to each Shimmer’s constructor[[8]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L250-L258). This Handler could run on a background thread (to offload work from the UI). We can create a dedicated HandlerThread (Android utility) for Shimmer data. This way, all device messages queue into this background thread’s Looper and are processed serially. - Within the Handler’s handleMessage, we will identify which device the data came from (the ObjectCluster likely contains the device MAC or an identifier) and route accordingly. The Shimmer API sets the msg.obj with an ObjectCluster that can provide the MAC address of the originating device[[18]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L194-L201). Using that, we can append the data to the correct file and also prepend a device ID if sending to PC in one stream.

**Kotlin Coroutines (alternative):** If using coroutines, we could achieve a similar structure: - Launch a coroutine for each device connection that reads data from the Shimmer’s input stream suspendingly. However, given we rely on Shimmer’s API, it’s easier to stick with their callback approach. - We might still use coroutines for other tasks (e.g., a coroutine to periodically check connection health, or to handle UI updates). - But for simplicity, using the HandlerThread pattern is sufficient and aligns with Shimmer’s design.

**Parallel Operations:**

* **Device Threads:** Each Shimmer device’s low-level I/O is already on its own thread (inside Shimmer class). So reading from multiple devices truly happens in parallel. The data just gets queued to our Handler.
* **Logging and Streaming Threads:** We should separate concerns such that writing to files or network doesn’t stall data reception:
* As mentioned, file I/O could be done right in the Handler (sequentially). If the volume is high, an alternative is to have another thread dedicated to disk writes. For example, use an ExecutorService with a single-thread executor for file writing, and post tasks to it from the Handler. This decouples disk latency from blocking the Handler.
* Similarly, network streaming can be handled by a thread or coroutine that consumes data from a queue. For instance, the Handler can push each sample (or batch of samples) into a ConcurrentLinkedQueue; a separate NetworkSenderThread loops and sends them over the socket.
* **Synchronization:** Use proper locks or concurrent structures if multiple threads share data. E.g., if the network thread reads from the same data objects the Handler is writing, make a copy or use thread-safe queues.

**UI Thread:** Keep heavy operations off the main thread: - Starting and stopping recordings can be triggered from UI, but those calls in ShimmerRecorder should quickly delegate to background work (e.g., initiating connections or stopping them). - If UI updates are needed (like showing live sensor values or device status), the HandlerThread can communicate with the main thread via a different Handler or LiveData/MutableStateFlow (if using Android architecture components). For example, if we get a new sensor reading and want to update a chart on screen, post that update to the UI main thread.

**Coroutines Example:** If using Kotlin, one might structure it as:

CoroutineScope(Dispatchers.IO).launch {  
 shimmerDevice.startStreaming() // non-blocking call  
 for(sample in shimmerDevice.sampleFlow) {   
 // hypothetically if Shimmer provided a Flow of samples  
 processSample(sample)  
 }  
}

But since Shimmer doesn’t natively provide a Flow, we adapt its callback to our own coroutine by emitting from the Handler.

In conclusion, the threading model should ensure **each Shimmer device is handled independently**, and logging/streaming occur without delaying one another. A HandlerThread processing all devices’ data sequentially is acceptable given the likely sample rates, but careful architecture (possibly one thread per device for processing) can further parallelize the workload if needed. The design should be robust enough that adding more devices linearly scales the load across threads.

*(The Shimmer devices deliver data via asynchronous messages*[*[2]*](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L150-L158)*, which we handle on a background Handler thread to avoid blocking the UI. This allows concurrent logging and streaming operations.)*

## 8. SessionInfo Integration (File Paths & Lifecycle)

The SessionInfo component coordinates the overall recording session, including metadata (session name, participant, start time) and file management for various recorders (RGB, thermal, etc.). Integrating ShimmerRecorder with SessionInfo ensures that:

* **File Management:** We obtain from SessionInfo a dedicated directory or filename pattern for storing Shimmer data logs. For example, sessionInfo.getSessionDirectory() might return a folder path like /storage/emulated/0/Recordings/Session\_001/. The ShimmerRecorder will create its CSV files in this folder. SessionInfo could also provide a unique session ID or timestamp to include in filenames to avoid collisions.
* **Lifecycle Hooks:** When a new session starts, SessionInfo (or a session controller) will call shimmerRecorder.startRecording(sessionInfo). This triggers device connection (if not already connected), sensor configuration, and then streaming + logging. Similarly, when the session stops, SessionInfo invokes shimmerRecorder.stopRecording(). We ensure that these methods perform the necessary cleanup:
* On start: use SessionInfo to perhaps record the exact session start time (for timestamp alignment), create new log files, and maybe write headers or session metadata. We might log in a file’s header some info like “Session ID, Device name, sampling rate, etc.” for traceability.
* On stop: finalize and close all file streams, annotate any EOF or summary if needed, and update SessionInfo with pointers to the saved files (SessionInfo might keep a list of data file paths for that session).
* **Concurrent Start with Other Recorders:** SessionInfo likely orchestrates that when a recording begins, all modalities start nearly simultaneously. We must ensure ShimmerRecorder is ready to stream when cameras start:
* One strategy: Establish Bluetooth connections to Shimmers **before** the session countdown or start trigger (to avoid delay). For instance, have the devices connected and in an idle state (not streaming) while waiting for the user to press "Start Session". Then on start, immediately send the startStreaming command to all devices. This reduces startup latency.
* Alternatively, if connection itself is quick, it can be done at start time, but Bluetooth connections can sometimes take a second or two, which could offset data relative to video.
* Integration wise, we might have SessionInfo call a prepare() on ShimmerRecorder in advance to connect devices, then call startRecording() at the moment of actual data capture. Prepare could also verify configurations.
* **Data Alignment:** SessionInfo might hold a master timestamp for session start (perhaps when the first frame of video is captured). We should use this to align Shimmer timestamps as described in section 6. For example, SessionInfo could provide sessionStartEpochMs. We then compute each sample’s absolute time = sessionStartEpochMs + (sampleTimestamp - t0). Here t0 is the device timestamp at session start (recorded in prepare).
* **File Path Example:** SessionInfo might tell us to save Shimmer logs as Session\_001\_Shimmer\_<devicename>.csv. We implement that naming. In case device name has characters not suitable for files (like colons in MAC), sanitize it (e.g., use last 4 digits of MAC or a user-defined alias).
* **Metadata Logging:** It could be useful to have SessionInfo’s summary include that Shimmer data was captured, how many devices, and where files are. Possibly, SessionInfo writes a JSON or XML manifest of the session. ShimmerRecorder should supply its part of that info (like device IDs, sensor types recorded, file names).
* **User Interface:** Through SessionInfo or the main UI, ensure that the state of ShimmerRecorder is reflected (e.g., if a device disconnects mid-session, the UI could show a warning via SessionInfo’s status).

By tightly integrating with SessionInfo, we guarantee that Shimmer recording aligns with the **session lifecycle** – it starts and stops in sync with other modalities, uses the correct file storage, and cleans up properly to avoid resource leaks (like Bluetooth connections left open).

*(SessionInfo coordinates file paths and session timing, so ShimmerRecorder will use it to obtain the output directory and to synchronize start/stop timing with the RGB and Thermal recorders. All Shimmer data files will be stored in the session folder for easy management.)* (ⓘ *No direct external citation for SessionInfo, as it is an internal component.*)

## 9. Resilience to Disconnection and Reconnection

When dealing with live wireless sensors, disconnections can happen (e.g., battery dies, out of range, Bluetooth interference). The system must be resilient to these events, ideally recovering without stopping the entire session.

**Detection of Disconnection:** The Shimmer API will notify via the Handler when a device’s state changes. Specifically, a message with ShimmerBluetooth.MSG\_IDENTIFIER\_STATE\_CHANGE and state=DISCONNECTED is sent if a device drops[[19]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L229-L237). The Handler can catch this: - Identify which device (the message likely includes the MAC or Shimmer object that disconnected[[18]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L194-L201)). - Log the event (e.g., print a message in the log file like “-- DISCONNECTED --” with a timestamp). - Update UI/SessionInfo so the user knows one sensor is down.

**Automatic Reconnection Strategy:** - We may attempt an automatic reconnect a few times. Immediately upon disconnect, the ShimmerRecorder can try to re-initiate connection to that device: - Call shimmer.connect(mac, ...) again or use btManager.connectShimmerThroughBTAddress(mac) if using the manager. - Possibly, first call btManager.disconnectShimmer(mac) to ensure any remnants are cleared (the Shimmer API connection test does something similar before reconnecting)[[20]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerConnectionTest/src/main/java/shimmerresearch/com/shimmerconnectiontest/MainActivity.java#L131-L139)[[1]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerConnectionTest/src/main/java/shimmerresearch/com/shimmerconnectiontest/MainActivity.java#L360-L365). - Limit the number of retries to avoid endless loops. For example, try up to N reconnection attempts with a short delay in between. N could be 3 or a user-configurable number. If all fail, mark the device as “offline” for the remainder of the session. - While reconnecting, continue recording from other devices uninterrupted. The architecture already handles each device independently, so a disconnect of one should not affect the others (aside from maybe logging a sync marker if needed).

**Data Continuity:** - If a device successfully reconnects, it may either resume streaming automatically or we may have to call startStreaming() again. The Shimmer API might require re-sending the sensor configuration after a reconnect (unless it’s preserved). We should reconfigure the sensors and restart streaming as part of the reconnection logic. - The data from before and after the dropout will be in the same file (which is simplest). There will be a gap in timestamps during the downtime. This is acceptable – it can be handled in analysis by looking at the timestamps. - Optionally, write a marker line in the CSV when disconnection happens and when reconnection occurs (to facilitate analysis). For example, a line with a special note: “DISCONNECTED at 12:05:30, reconnecting…”.

**User Notification:** - The app should alert the user if a sensor disconnects. Possibly through the UI (e.g., a toast or a warning icon next to the device name). - If reconnection is successful, notify the user (“Shimmer A reconnected”). If not, inform that data from that device is lost.

**Edge Cases:** - If a device repeatedly disconnects (flaky connection), decide whether to keep retrying or stop. Possibly after several fails, give up to avoid disrupting the session. - If a device disconnects very close to session end, the user might just stop the session rather than wait for reconnection. - Ensure that the stopRecording() still works even if a device is mid-reconnect attempt or offline. The stop routine should attempt to disconnect any remaining connections (even offline ones can be marked as closed).

By implementing these measures, the system will be robust against interruptions. The session won’t need to be fully restarted if one sensor drops out briefly. Instead, it will attempt to self-heal and continue recording synchronized data from all available sources.

*(The Shimmer driver reports a state change when a device disconnects*[*[19]*](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L229-L237)*. We leverage that to trigger reconnection logic, ensuring the session can continue with minimal interruption.)*

## 10. Manual Test Plan

To verify the Shimmer3 GSR+ integration meets all requirements, a comprehensive manual test should be performed:

1. **Initial Setup & Pairing:**
2. Charge at least two Shimmer3 GSR+ devices and ensure their Bluetooth is on.
3. On the Android device, if not already paired, open system Bluetooth settings and pair with each Shimmer (enter PIN 1234 when prompted). Alternatively, test in-app pairing:
   * Launch the app and navigate to the Shimmer device setup screen.
   * If permissions dialog appears, grant Bluetooth and location permissions.
   * Tap “Scan for Shimmer Devices”. Confirm a list of available devices appears. Select one device, pair it (enter PIN) if needed. Then repeat to add the second device.
   * Verify that both devices show up as “paired/selected” in the app’s UI.
4. **Expected:** The app successfully obtains the MAC addresses of both Shimmers with no errors. Devices should indicate they are paired (e.g., LED behavior or in phone’s paired devices list).
5. **Permission Handling:**
6. If on Android 12+, ensure the app requested BLUETOOTH\_SCAN/CONNECT properly. Try denying and see if the app gracefully handles (it should prompt again or show a message).
7. Turn off Bluetooth and start the app – it should prompt to enable Bluetooth. Enable it and continue.
8. **Connecting Multiple Devices:**
9. In the app, tap “Connect” or start the session (depending on implementation) to connect to the selected Shimmers.
10. **Expected:** The app connects to each device in turn. Perhaps an on-screen status changes from “Not Connected” to “Connected” for each device. No crashes or hangs during this process.
11. Both devices’ LEDs on the Shimmer (if applicable) might indicate connection (often Shimmer has a LED that flashes when streaming).
12. If one device fails to connect (e.g., if it’s off), the app should notify and allow retry or continue with the other device.
13. **Channel Selection:**
14. The app should display channel toggles for each device. Toggle various combinations:
    * For device A, enable GSR and PPG, disable accelerometer.
    * For device B, enable all channels (Accel, GSR, PPG, etc.).
15. Start streaming (but not yet recording to file, if separate step) to see if data comes through:
    * Perhaps the app has a debug view or prints logcat messages for incoming data. Check that only the enabled channels produce data. E.g., for device A we expect GSR and PPG values, for device B we expect a full set including accel.
    * If possible, use Shimmer’s own tools (like Shimmer’s Consensys software or another app) as a cross-check for what the device is sending.
16. **Expected:** Channel configuration is respected – no data from disabled channels. The system should handle enabling/disabling without disconnecting.
17. **Start Recording Session (Integration with Cameras):**
18. Initiate a new recording session that involves RGB video, thermal recording, and Shimmer data:
    * Enter session metadata in UI (if any, like subject ID).
    * Start the session. All recorders (RGB, thermal, Shimmer) should commence.
19. Clap hands or perform a distinct action in front of the RGB camera while simultaneously triggering a response on Shimmer (e.g., briefly touch GSR electrodes to cause a spike, or move the device if accelerometer is on). This creates a synchronizing event across modalities.
20. Let the session run for a few minutes. Observe that the app remains responsive (no freezes).
21. **Real-time streaming to PC:** On the PC, run the companion software or a socket listener:
    * Confirm that the PC is receiving data from the phone. E.g., lines of sensor data appear in the PC app/console.
    * Check that data from both Shimmer devices is coming through (perhaps tagged by device).
    * Induce a noticeable change: for instance, physically shake one Shimmer (to change accel) or breathe on GSR sensor to change readings, and see that change on PC in real-time.
22. While streaming, also check the phone UI if it shows any live values (not required, but some apps display current sensor values). Ensure they update for both devices.
23. **Disconnection and Reconnection:**
24. During the session, turn off one Shimmer device (simulating battery loss or range drop).
    * The app should log the disconnection event. It might show a warning in UI.
    * The PC stream should indicate that device’s data stopped or a disconnect notice if implemented.
    * The other device should continue streaming uninterrupted.
25. Turn the device back on within a short time. The app should attempt to reconnect (if auto-reconnect is implemented):
    * Expect to see a “reconnected” message or the device status going back to streaming.
    * The data for that device should resume logging and streaming to PC. Verify that new data points (after reconnection) are appended in the correct log file (not a new file).
26. If auto-reconnect isn’t implemented or fails, try manually tapping a “reconnect” button if provided.
27. **Expected:** The session continues and the device either reconnects seamlessly or, if not, the failure is contained (other data still recording, and session doesn’t crash). The moment of disconnect and reconnect is clearly recorded.
28. **Stop Recording:**
29. End the session via the UI stop button.
30. Ensure all Shimmer devices stop streaming. The Shimmer devices might have an LED that turns off or they might go idle. The app should close the Bluetooth connections.
31. Confirm the PC stops receiving data (maybe an end-of-stream indication or simply no more data).
32. After stopping, the app might present a summary or make the recorded files available.
33. **Data Verification on Device:**
34. Using a file browser or by connecting the phone to a PC, navigate to the saved session files. There should be CSV files (or chosen format) for each Shimmer device from the session.
35. Open the files and check:
    * All columns present with headers.
    * Data looks plausible (numeric values in expected ranges, e.g., GSR around a few microsiemens, PPG values oscillating, accel ~9.8 for gravity on one axis when static, etc.).
    * Timestamps increasing monotonically. Verify if the timestamps line up with session time:
    * The first timestamp should correspond to session start time (compare with video start).
    * If an event like the hand clap was at 00:01:23 in video, see if there’s a sensor change at roughly the same timestamp in the sensor logs.
    * The file should cover the entire duration of the session (start to stop).
    * Check that when the device was turned off, either there’s a gap or a marker in the data, and after turning on, data resumes (with or without a timestamp jump).
36. **Data Verification on PC:**
37. If the PC app logs data, save that and compare with phone log:
    * The data received live should match the data in the phone’s log for the same timestamps. (This ensures streaming was accurate.)
    * Minor differences in timing (a few milliseconds) are okay, but sequence of values should be the same.
38. **Multi-Device Synchronization:**
    * If possible, cross-compare the two Shimmer devices’ logs. Because they started together, their timestamps should be in the same timebase (e.g., if both use epoch ms, at any given real time the timestamps are close).
    * Check that a simultaneous event (like both devices experiencing a jolt if they were moved together) appears at the same timestamp in both files.
39. **Resource Cleanup:**
    * Start another session after the previous without restarting the app:
    * The app should allow connecting the same devices again (or they might still be connected). If they were left connected, ensure the second session can start streaming immediately.
    * No issues like “device already in use” or multiple connections to same device should occur.
    * Also try disconnecting devices after stop: perhaps turn off a Shimmer after session and see the app handles it (since session ended, it might just silently drop the connection).
    * Exit the app and ensure no stray Bluetooth connections remain (the Shimmers should not still show as connected once app is closed).

**Expected Results:** All tests above should pass – devices connect and stream concurrently, data is logged and streamed correctly with proper timestamps, and the system handles disconnects gracefully. The data collected should be synchronized with other modalities (within a small margin of error). The app’s performance should remain smooth, indicating our threading model is effective (e.g., no significant UI lag even with two Shimmers streaming and file I/O).

By following this test plan, we can validate that the implementation meets the requirements for **modularity, scalability (multi-device)**, and **accurate synchronized capture** of Shimmer3 GSR+ sensor data alongside the existing recording system. All issues discovered during testing should be addressed before considering the milestone complete.

[[1]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerConnectionTest/src/main/java/shimmerresearch/com/shimmerconnectiontest/MainActivity.java#L360-L365) [[20]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerConnectionTest/src/main/java/shimmerresearch/com/shimmerconnectiontest/MainActivity.java#L131-L139) MainActivity.java

<https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerConnectionTest/src/main/java/shimmerresearch/com/shimmerconnectiontest/MainActivity.java>

[[2]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L150-L158) [[4]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L53-L61) [[5]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L79-L87) [[7]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L127-L134) [[8]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L250-L258) [[14]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L166-L174) [[15]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L170-L178) [[18]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L194-L201) [[19]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java#L229-L237) MainActivity.java

<https://github.com/ShimmerEngineering/ShimmerAndroidAPI/blob/d9483d025802ec9378df8d4d863c763ffc655411/ShimmerAndroidInstrumentDriver/shimmerBasicExample/src/main/java/shimmerresearch/com/shimmerbasicexample/MainActivity.java>

[[3]](https://www.shimmersensing.com/product/shimmer-java-android-api/#:~:text=,Auto%20calibration%20of%20data) [[9]](https://www.shimmersensing.com/product/shimmer-java-android-api/#:~:text=,Auto%20calibration%20of%20data) Shimmer Java/Android API - Shimmer Wearable Sensor Technology

<https://www.shimmersensing.com/product/shimmer-java-android-api/>

[[6]](https://shimmersensing.com/wp-content/docs/support/getting-started/Streaming_to_Android.pdf#:~:text=The%20PIN%20is%201234,PAIRING%20THE%20SHIMMER%20OVER%20BLUETOOTH) Getting Started - Android

<https://shimmersensing.com/wp-content/docs/support/getting-started/Streaming_to_Android.pdf>

[[10]](https://github.com/amaltesh/shimmer/blob/6181b187c489adec57fd0cb293b176da6aa4569a/ShimmerAndroidInstrumentDriver/src/com/shimmerresearch/android/Shimmer.java#L2-L10) [[11]](https://github.com/amaltesh/shimmer/blob/6181b187c489adec57fd0cb293b176da6aa4569a/ShimmerAndroidInstrumentDriver/src/com/shimmerresearch/android/Shimmer.java#L26-L34) Shimmer.java

<https://github.com/amaltesh/shimmer/blob/6181b187c489adec57fd0cb293b176da6aa4569a/ShimmerAndroidInstrumentDriver/src/com/shimmerresearch/android/Shimmer.java>

[[12]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/wiki#:~:text=In%20the%20Android%20API%2C%20you,to%20perform%20the%20same%20function) [[13]](https://github.com/ShimmerEngineering/ShimmerAndroidAPI/wiki#:~:text=The%20differences%20between%20these%20methods,are%20as%20follows) Home · ShimmerEngineering/ShimmerAndroidAPI Wiki · GitHub

<https://github.com/ShimmerEngineering/ShimmerAndroidAPI/wiki>

[[16]](https://dev.to/techkoool/how-to-transmit-android-real-time-sensor-data-to-computer-5308#:~:text=private%20class%20WifiAsyncTask%20extends%20AsyncTask,Socket%20socket%3B%20private%20PrintWriter%20printWriter) [[17]](https://dev.to/techkoool/how-to-transmit-android-real-time-sensor-data-to-computer-5308#:~:text=1,connection%20using%20the%20PrintWriter%20object) How to transmit android real-time sensor data to computer? - DEV Community

<https://dev.to/techkoool/how-to-transmit-android-real-time-sensor-data-to-computer-5308>