# Implementation Guide for Android Multi-Modal Recording App (Milestone 2.1)

## Goals and Design Principles

The goal of Milestone 2.1 is to implement the **Android phone application** for a synchronized multi-modal recording system. This app will capture **4K RGB video (with RAW image support)**, **thermal IR video** (via the Topdon SDK), and **Shimmer sensor data** simultaneously, while supporting **real-time preview streaming** and a **socket interface** for external control. Key design principles include:

* **Foreground Service for Long Recording:** Use a **foreground RecordingService** to manage recording outside the UI thread, ensuring the system knows recording is in progress (with a persistent notification) and preventing the service from being killed mid-recording[[1]](https://developer.android.com/develop/background-work/services#:~:text=A%20foreground%20service%20performs%20some,isn%27t%20interacting%20with%20the%20app). This service must be lifecycle-aware and coroutine-friendly, meaning it will cleanly handle coroutine scope cancellation on stop.
* **Modularity & Clean Architecture:** Separate concerns into distinct components – e.g. CameraRecorder for RGB camera, ThermalRecorder for IR camera, ShimmerRecorder for sensor streaming, SocketController for network communication, and PreviewStreamer for handling live previews. Each component has a single responsibility, making the system extensible and maintainable. These modules interact via well-defined interfaces or callbacks (e.g. notifying the service when recording is started/stopped or when errors occur). This aligns with Clean Architecture principles (separation of concerns, low coupling).
* **Concurrency and Synchronization:** Leverage Kotlin **coroutines** to run recording tasks concurrently (camera capture, sensor read, network send, etc.) without blocking the main thread. Ensure all blocking or heavy operations are offloaded from the UI thread to background coroutine dispatchers (to avoid ANRs)[[2]](https://developer.android.com/develop/background-work/services#:~:text=Caution%3A%20A%20service%20runs%20in,ANR%29%20errors). Use synchronized timestamps or start triggers to keep modalities in sync (e.g. mark the start time and include timestamps in sensor data).
* **High Performance & Hardware Utilization:** Use hardware-accelerated codecs and camera capabilities. For example, use the Camera2 API (or CameraX if feasible) at **FULL/LEVEL\_3** capability to allow simultaneous outputs including RAW capture[[3]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Camera%20Capture%20,RAW%20images%20at%20intervals%20during). Record 4K video using **MediaRecorder/MediaCodec** for efficient H.264/H.265 compression (offloading to hardware encoders)[[4]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=High,A%20compromise), and handle thermal frames appropriately. Minimize performance overhead for preview streaming (e.g. use a lower-resolution feed for preview to avoid stalling the 4K recording[[5]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=standard,tested%20on%20the%20specific%20hardware)).
* **Extensibility and Future Integration:** Design the app to work standalone (local start/stop from the UI) for initial testing, but **future-proof** it for external PC control via sockets. The socket interface should be designed to easily extend with new commands (start/stop, calibration trigger, etc.). By isolating the SocketController, we can later integrate PC synchronization logic without major changes to core recording modules.
* **Reliability and Logging:** Include robust error handling and logging. The service should handle edge cases (camera errors, sensor disconnects, low storage) gracefully – e.g. stop recording and notify the user if an error occurs. Implement a logging mechanism (Logger) to tag and record significant events (to logcat and optionally to a file) for debugging and verification. This will help in a research setting to diagnose issues (e.g. if Phone2 preview isn’t showing or sensor data flatlines).

## Project Structure and Package Layout

To organize the code, we propose a clear package structure within the Android Studio project, separating UI, service, and feature modules. For example:

com.example.multirecorder/ (Application package)  
├─ ui/ (UI layer)  
│ └─ MainActivity.kt (Activity with start/stop controls, previews)  
├─ service/ (Foreground service and session mgmt)  
│ ├─ RecordingService.kt (Foreground Service orchestrating recording)  
│ └─ SessionManager.kt (Manages session folder and file naming)  
├─ recording/ (Recording modules for each data source)  
│ ├─ CameraRecorder.kt (Handles RGB camera capture & 4K video/RAW)  
│ ├─ ThermalRecorder.kt (Handles IR camera via Topdon SDK)  
│ └─ ShimmerRecorder.kt (Handles Shimmer sensor Bluetooth stream)  
├─ comms/ (Communication modules)  
│ ├─ SocketController.kt (Manages socket connection and commands)  
│ └─ PreviewStreamer.kt (Streams preview frames to PC or UI)  
├─ util/ (Utility classes and helpers)  
│ ├─ Logger.kt (Logging helper)  
│ └─ FileUtils.kt (File I/O utilities, e.g., directory creation)  
└─ (possibly other packages as needed, e.g., model/ for data classes)

This layout ensures a **modular structure**. The RecordingService in service acts as the central coordinator. Under recording, each recorder class focuses on one modality (camera, thermal, shimmer). The comms package contains networking (socket) and preview streaming logic, which could be toggled on/off as needed. The util package holds generic helpers (for logging, file management, etc.).

Each module can be developed and tested in isolation. For example, you could unit-test SessionManager naming logic without launching the whole app, or test ShimmerRecorder with a dummy data source. This structure also makes it easier to scale (e.g., adding a new sensor module in the future or swapping the communication method to MQTT) with minimal impact on other components.

## Gradle Build Configuration and Dependencies

Setting up the Gradle build is crucial to include all required libraries and ensure compatibility with camera and sensor APIs:

* **Compile SDK and Language:** Set compileSdk (and targetSdk) to the latest Android API (at least 33 or 34) to leverage modern APIs (CameraX concurrency, new Bluetooth permissions, etc.). Use Kotlin with Java 1.8 (or higher) compatibility for coroutines and language features. For example, in build.gradle (Module: app):

android {  
 compileSdkVersion 34  
 defaultConfig {  
 applicationId "com.example.multirecorder"  
 minSdkVersion 26 // e.g., Camera2 and BLE require at least API 21; use 26+ if possible  
 targetSdkVersion 34  
 versionCode 1  
 versionName "1.0"  
 }  
 compileOptions {  
 sourceCompatibility JavaVersion.VERSION\_1\_8  
 targetCompatibility JavaVersion.VERSION\_1\_8  
 }  
 kotlinOptions { jvmTarget = "1.8" }  
}

* **Core AndroidX Libraries:** Include standard AndroidX dependencies for compatibility and UI:
* implementation "androidx.core:core-ktx:1.10.1"  
  implementation "androidx.appcompat:appcompat:1.6.1"  
  implementation "com.google.android.material:material:1.9.0"
* These provide base support (Material buttons, etc.). Also add constraintlayout if using it in UI layout.
* **Camera and Media:** Since we plan to use Camera2 API directly (for fine-grained control over RAW and multi-output), we don’t need a specific CameraX dependency. The Camera2 classes are part of the Android SDK. If desired, you could add CameraX for ease of preview, but CameraX may not yet fully support multi-camera RAW scenarios. We will proceed with Camera2. If using CameraX for simpler devices, you’d add:
* implementation "androidx.camera:camera-core:1.2.3"  
  implementation "androidx.camera:camera-camera2:1.2.3"  
  implementation "androidx.camera:camera-lifecycle:1.2.3"
* (But for 4K + RAW, Camera2 is the better approach).
* **Topdon IR SDK:** The **Topdon thermal camera SDK** likely comes as a .aar or .jar library. Since you have a topdon-sdk repository, you can include it as a module or simply drop the provided library into your app. For example, if they provided libIRSDK.aar, place it in app/libs/ and add in Gradle:
* implementation files('libs/ANDROID\_IR\_SDK.aar')
* (Assuming that .aar contains the necessary classes like UVCCamera, etc.). The Topdon SDK may also depend on **USB host features** (it likely wraps libuvc). Make sure to enable USB host mode in the manifest (see Manifest section). The sample Topdon app uses a USBMonitor to handle device connection; by including the SDK .aar and any native libs, our ThermalRecorder can call into it.
* **Shimmer API:** The Shimmer Android API can be integrated either by including its source as a library module or via a JitPack/Maven artifact if available. If Shimmer provides a Maven dependency (check their documentation), it might be something like:
* implementation 'com.shimmer:sensor-api:3.0.0'
* (Placeholder – actual coordinates might differ; Shimmer’s GitHub suggests downloading the API). Alternatively, clone the ShimmerAndroidAPI repo and add it as a module or include the relevant .jar from their release. Ensure the Bluetooth permissions are handled (discussed in Manifest). The Shimmer API will provide classes for scanning and connecting to Shimmer3 devices[[6]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Shimmer%20Integration%3A%20Utilize%20the%20Shimmer,that%20we%20want%20minimal%20delay).
* **Kotlin Coroutines:** Add coroutines for background tasks:
* implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.7.1"  
  implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.1"
* This gives us Dispatchers.IO for I/O threads, etc., and Android-specific support to tie into lifecycles.
* **Logging Library (Optional):** For improved logging, you could use a library like **Timber**:
* implementation "com.jakewharton.timber:timber:5.0.1"
* Timber makes tagged logging easier. However, using Android’s built-in Log is fine for simplicity. We will implement a Logger util that can wrap either.
* **Other Dependencies:** If you plan on JSON messaging over the socket (for commands), you might include a JSON library like Kotlinx Serialization or Gson:
* implementation "com.google.code.gson:gson:2.10.1"
* This can help parse incoming commands like {"cmd":"START\_RECORD"} easily. Similarly, if using WebSockets instead of raw TCP, you might add OkHttp’s WebSocket or a library – but to keep things simple, a standard Socket will do.
* **Android Studio Config:** Ensure **NDK** is configured if the Topdon SDK uses native libraries (the sample gradle indicated NDK filters for ABIs[[7]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/build.gradle#L18-L26)). In our project, if the .aar includes .so files for different ABIs, configure ndk { abiFilters "armeabi-v7a", "arm64-v8a" } (and others as needed) to package the correct native libs. Also, enable **ViewBinding** or **DataBinding** if you prefer for the UI (the sample had viewBinding enabled[[8]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/build.gradle#L44-L51)). This can help manage the preview TextureViews in MainActivity.

After adding these, do a Gradle sync in Android Studio. The project should build, and you’ll have access to all required APIs. Verify that the external SDKs (Topdon, Shimmer) are recognized by the IDE (sometimes .aar needs implementation fileTree(dir: "libs", include: ["\*.aar"]) which we did, or adding flatDir repositories).

## Android Manifest and Required Permissions

The AndroidManifest.xml must declare all needed permissions and components. Here’s a checklist of what to include:

**App Components:**

* **Foreground Service declaration:** Add an entry for RecordingService. For example:
* <service  
   android:name=".service.RecordingService"  
   android:exported="false"  
   android:foregroundServiceType="camera|microphone|location">  
   <!-- No intent-filter needed; we start it explicitly -->  
  </service>
* Setting exported="false" ensures no other app can start this service. We specify relevant foregroundServiceType flags for Android 10+ if needed: we use camera|microphone if recording audio, and possibly location if we ever tag sensor data with location or if Bluetooth scanning is considered “Nearby devices” (not exactly location, see below). This is future-proofing; the system requires type for certain use-cases on Android 14+ (e.g., camera type ensures we have CAMERA permission when service runs)[[9]](https://developer.android.com/about/versions/14/changes/fgs-types-required#:~:text=FOREGROUND_SERVICE_TYPE_CAMERA%20Runtime%20prerequisites)[[10]](https://developer.android.com/about/versions/14/changes/fgs-types-required#:~:text=match%20at%20L755%20FOREGROUND_SERVICE_TYPE_MICROPHONE%20Runtime,prerequisites).
* **USB host features (for external camera):** Since the IR camera is an external USB device, declare the USB host feature:
* <uses-feature android:name="android.hardware.usb.host" android:required="true"/>
* This lets Google Play (or the system) know the app uses USB host mode (meaning it expects a device with USB OTG). Also, declare camera features:
* <uses-feature android:name="android.hardware.camera" android:required="true"/>
* (Our app definitely needs a camera.) If the IR camera were internal, perhaps <uses-feature android:name="android.hardware.camera.ir" ...> if such exists, but since it’s external, not needed.
* **USB Device Intent Filter:** To automatically detect and get permission for the IR camera when plugged in, include in the manifest (likely under an activity or the service):
* <intent-filter>  
   <action android:name="android.hardware.usb.action.USB\_DEVICE\_ATTACHED" />  
  </intent-filter>  
  <meta-data android:name="android.hardware.usb.action.USB\_DEVICE\_ATTACHED"  
   android:resource="@xml/device\_filter" />
* This listens for the USB device attach event[[11]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/AndroidManifest.xml#L28-L36). You’ll need a device\_filter.xml (placed in res/xml/) specifying the vendor and product ID of the Topdon camera (the SDK documentation or example should provide these). This filter allows the system to auto-launch our app (or at least notify it) when the device connects, and it streamlines permission granting for USB. In our case, since the app likely will be running already, the USBMonitor in the SDK can also request permission. But having the filter ensures we declare interest in the device.

**Permissions:**

* **Camera and Audio:**
* <uses-permission android:name="android.permission.CAMERA" /> – needed for Camera2 (runtime permission as well).
* <uses-permission android:name="android.permission.RECORD\_AUDIO" /> – if we capture microphone audio with the video (likely we should, to record participant audio or any sound; if audio isn’t needed, omit it). If included, also add foregroundServiceType="microphone" in service as above.
* **Storage:**
* Since we’ll save video and data files, request write permissions. For Android 10 and below: <uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />. For Android 11+, that alone isn’t sufficient; you either use the scoped storage (MediaStore API) or declare the broad MANAGE\_EXTERNAL\_STORAGE. In a research context (and given the sample’s approach), we can use the legacy approach: in manifest set android:requestLegacyExternalStorage="true" in the application tag (as seen in Topdon sample[[12]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/AndroidManifest.xml#L10-L18)), and include:
* <uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE" />  
  <uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"  
   tools:ignore="ScopedStorage" />
* Additionally, for Android 11+, you might declare:
* <uses-permission android:name="android.permission.MANAGE\_EXTERNAL\_STORAGE"  
   tools:ignore="ProtectedPermissions" />
* This permission is special (requires user to enable in settings). If feasible, consider saving recordings in app-specific storage (which doesn’t require runtime permission) or use the Storage Access Framework to write to DCIM. For simplicity, we can stick to the legacy storage with user granting WRITE permission at runtime. The files can be large, so ensure the target directory is not on low-storage internal storage if possible (maybe use external SD if available or large internal).
* **Bluetooth:**  
  The Shimmer connects via Bluetooth. On modern Android (12+), we have new permissions:
* <!-- Legacy permissions for older devices: -->  
  <uses-permission android:name="android.permission.BLUETOOTH" android:maxSdkVersion="30"/>  
  <uses-permission android:name="android.permission.BLUETOOTH\_ADMIN" android:maxSdkVersion="30"/>  
  <!-- New permissions for Android 12+ : -->  
  <uses-permission android:name="android.permission.BLUETOOTH\_SCAN" />  
  <uses-permission android:name="android.permission.BLUETOOTH\_CONNECT" />
* If the Shimmer uses BLE scanning, we include BLUETOOTH\_SCAN (and possibly ACCESS\_FINE\_LOCATION because BLE scans can infer location, unless we add the neverForLocation flag)[[13]](https://developer.android.com/develop/connectivity/bluetooth/bt-permissions#:~:text=If%20your%20app%20targets%20Android%C2%A012,in%20your%20app%27s%20manifest%20file)[[14]](https://developer.android.com/develop/connectivity/bluetooth/bt-permissions#:~:text=%3C%21,permission%20android%3Aname%3D%22android.permission.BLUETOOTH_ADVERTISE%22). If it’s classic Bluetooth and the Shimmer is already paired, we might only need BLUETOOTH\_CONNECT to communicate. We will request these at runtime in the app (they are runtime dangerous permissions on Android 12+). Also, if Shimmer requires discoverability or advertising (unlikely for our use), BLUETOOTH\_ADVERTISE would be needed – but typically the phone is the central, so SCAN (to find devices) and CONNECT are key.  
  *Note:* Starting a Bluetooth scan on new Android triggers a system “Nearby devices” dialog for the user to approve[[15]](https://developer.android.com/develop/connectivity/bluetooth/bt-permissions#:~:text=The%20,as%20shown%20in%20figure%201). We should guide the user to allow it when connecting the sensor.
* **Internet/Network:**  
  For socket communication over Wi-Fi or any network, add:
* <uses-permission android:name="android.permission.INTERNET" />  
  <uses-permission android:name="android.permission.ACCESS\_WIFI\_STATE" />
* INTERNET is obvious for socket I/O. ACCESS\_WIFI\_STATE can be useful if we want to check if Wi-Fi is connected or get the IP of the PC, though not strictly required for socket use. (If using mobile hotspot or so, could also include ACCESS\_NETWORK\_STATE to be thorough.)
* **Wake Lock:**  
  To prevent the device from sleeping during long recordings:
* <uses-permission android:name="android.permission.WAKE\_LOCK" />
* The service can acquire a wake lock so CPU stays on (especially if screen is off during recording). Since we are a foreground service, the system already holds a partial wake lock while in foreground, but explicitly using WakeLock can give more control if needed. The Topdon sample included WAKE\_LOCK[[16]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/AndroidManifest.xml#L80-L84), which is a good practice.
* **Foreground Service Permission:**  
  From Android Pie (API 28) onward, apps must declare:
* <uses-permission android:name="android.permission.FOREGROUND\_SERVICE" />
* This is automatically granted (normal permission) but is required in the manifest to start a foreground service[[17]](https://stackoverflow.com/questions/52382710/permission-denial-startforeground-requires-android-permission-foreground-servic#:~:text=Permission%20Denial%3A%20startForeground%20requires%20,so%20the%20system%20automatically). So include that as well.
* **Other Sensors:** (If Shimmer sensor data included things like body sensors, you might need <uses-permission android:name="android.permission.BODY\_SENSORS" />, but since Shimmer is external and streams via Bluetooth, we don’t need the phone’s sensors permission.)

After declaring these in the manifest, at runtime we will need to **request** the dangerous ones (Camera, Audio, Location (if BLE), Bluetooth Scan/Connect, Storage) from the user via the Activity (using the ActivityCompat.requestPermissions). The app should handle the case where a user denies a permission – e.g., if CAMERA or WRITE storage is denied, we cannot proceed with recording. Since this is a lab app, we can assume we’ll guide the user (ourselves or lab operator) to grant all on first launch.

Double-check that the manifest also includes the app’s **application name** (if a custom Application class is used for global init, e.g., Topdon’s sample had android:name=".MyApplication"). If our app doesn’t need a custom Application subclass, we can skip that. But if the Topdon SDK requires some init (maybe not), we could initialize it in Application.onCreate.

Finally, verify the **device\_filter XML** for the IR camera: it should list the vendor and product ID of the camera. The Topdon SDK documentation or example likely provides this (for instance, a common FLIR one might have certain VID/PID). By matching that, Android will know to prompt permission when that device is connected. Otherwise, we’ll manually request via UsbManager.

## Core Modules and Class Design

Now we break down the core classes/modules and their responsibilities. Each class will be designed with thread-safe coroutine usage and a clear API. We’ll include sample method signatures and how they work together:

### **RecordingService** (Foreground Service)

**Role:** Orchestrates all recording components. It starts/stops the camera, thermal, and sensor recorders, manages the session folder, and keeps the foreground notification. It also interfaces with the SocketController (receiving external start/stop commands) and the UI (MainActivity can bind or send intents to control it).

**Lifecycle:** The RecordingService is a started service (we call startForegroundService to launch it). It runs in the **main thread** of the app’s process by default, so we will create a **CoroutineScope** in the service for concurrent tasks (or use something like LifecycleService from AndroidX which provides a Lifecycle owner). Because a service doesn’t automatically have a lifecycle like Activity, one approach is to extend LifecycleService so that it can have a Lifecycle and we can use lifecycleScope. Alternatively, we manually manage a CoroutineScope with job = SupervisorJob() and cancel it in onDestroy().

**Notification:** On start, the service will immediately promote itself to foreground with startForeground(notificationId, notification). The notification should have an ongoing icon (e.g., a “Recording…” text, maybe with a stop action button if we want). Android requires this for any long-running background operation[[1]](https://developer.android.com/develop/background-work/services#:~:text=A%20foreground%20service%20performs%20some,isn%27t%20interacting%20with%20the%20app).

**Communication:** We can allow both **intent commands** and **socket commands** to control the service: - *Intents:* MainActivity can send an explicit intent with action, e.g., "ACTION\_START\_RECORD" or "ACTION\_STOP\_RECORD". In onStartCommand, the service checks the intent action and acts accordingly. This decouples the UI button from service logic. - *Socket:* The SocketController (running in its own thread/coroutine) can invoke callbacks on the service when it receives commands from PC (e.g., if PC sends "START" message, SocketController calls a service.startRecording() method).

**Coroutine usage:** For example, when starting recording, the service might launch parallel coroutines: one for the camera recorder, one for thermal, one for shimmer. We would use something like:

val recordingJob = SupervisorJob()  
val scope = CoroutineScope(Dispatchers.Default + recordingJob)  
scope.launch { cameraRecorder.start(sessionInfo) }  
scope.launch { thermalRecorder.start(sessionInfo) }  
scope.launch { shimmerRecorder.start(sessionInfo) }

Using SupervisorJob means if one fails, others can continue or we can handle failure individually. We must also coordinate the start/stop so that all begin at roughly the same time (for sync, ideally within milliseconds). We could start them sequentially in quick succession, or if perfect sync is needed, instruct them to align on a timestamp (not needed at this stage, but possibly later with PC command containing a start time).

When stop is requested, we similarly call each recorder’s stop method (which should handle closing files/devices). We’d also cancel the recordingJob to cancel any remaining coroutines (e.g., preview streaming coroutine).

**Pseudo-code structure:**

class RecordingService : Service() {  
 private val notificationId = 1  
 private lateinit var cameraRec: CameraRecorder  
 private lateinit var irRec: ThermalRecorder  
 private lateinit var shimmerRec: ShimmerRecorder  
 private lateinit var session: SessionManager  
 private var socketCtrl: SocketController? = null  
  
 override fun onCreate() {  
 super.onCreate()  
 // Initialize components  
 session = SessionManager(applicationContext)  
 cameraRec = CameraRecorder(applicationContext)  
 irRec = ThermalRecorder(applicationContext)  
 shimmerRec = ShimmerRecorder(applicationContext)  
 socketCtrl = SocketController(/\* maybe pass callback \*/)   
 // Prepare foreground notification  
 startForeground(notificationId, createNotification("Idle"))  
 // Optionally, start socket listening  
 socketCtrl?.startListening { cmd -> handleSocketCommand(cmd) }  
 }  
  
 override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {  
 when(intent?.action) {  
 "ACTION\_START\_RECORD" -> beginRecordingSession()  
 "ACTION\_STOP\_RECORD" -> endRecordingSession()  
 }  
 return START\_STICKY  
 }  
  
 private fun beginRecordingSession() {  
 val sessionInfo = session.createNewSession() // create folder, filenames  
 // Update notification to "Recording..."  
 updateNotification("Recording session ${sessionInfo.name}")  
 // Launch recorders concurrently  
 recordingScope = CoroutineScope(Dispatchers.Default + SupervisorJob())  
 recordingScope.launch { cameraRec.startRecording(sessionInfo) }  
 recordingScope.launch { irRec.startRecording(sessionInfo) }  
 recordingScope.launch { shimmerRec.startRecording(sessionInfo) }  
 // Also perhaps start preview streaming in another coroutine  
 recordingScope.launch { previewStreamer.start(sessionInfo) }  
 }  
  
 private fun endRecordingSession() {  
 // Stop preview first to reduce load  
 previewStreamer.stop()  
 // Stop recorders (in parallel or sequence)  
 cameraRec.stopRecording()  
 irRec.stopRecording()  
 shimmerRec.stopRecording()  
 // Cancel any remaining tasks  
 recordingScope.cancel()  
 // Update notification to "Saved" or end foreground  
 updateNotification("Recording stopped")  
 stopForeground(true)  
 // Maybe stopSelf() if we want to terminate service after stopping  
 }  
  
 override fun onDestroy() {  
 super.onDestroy()  
 socketCtrl?.stop() // close socket if running  
 recordingScope.cancel() // ensure no coroutines leaking  
 }  
  
 ...  
}

This pseudo-code illustrates the flow. The actual implementation will include proper error handling (try/catch around recorder starts to handle exceptions, etc.). Note we call stopForeground(true) after stopping recording to remove the notification; we might keep service running if expecting another session, or we can stopSelf if done.

The service will use **SessionManager** (described below) to set up the directory and filenames for this run. It passes that info to each recorder so they know where to save their files.

### **SessionManager** (Session Folder & File Naming)

**Role:** Manages creation of a new session directory and standardized file naming. It encapsulates the naming scheme (timestamps, IDs, etc.) so other components don’t worry about paths.

When session.createNewSession() is called (e.g., at the start of each recording), it will: - Determine a base directory for all recordings. For example, use external storage directory: Documents/MultiRecorder or Movies/MultiRecorder. Alternatively, context.getExternalFilesDir(null) which gives an app-specific folder. - Create a new subdirectory for this session. Could be named with a timestamp, e.g., "Session\_2025-07-27\_19-38-00", or if there’s a participant or trial ID, include that. For uniqueness, timestamp down to seconds (or an index if multiple in same second). - Prepare filenames for each modality file. For instance: - rgb\_video.mp4 for main camera video, - ir\_video.mp4 for thermal camera video, - raw\_frames.dng or a folder of DNG images if RAW captures are taken (or perhaps we embed occasional RAW into the video as metadata; but simplest is to save separate images), - shimmer\_data.csv for sensor readings, - maybe a session\_meta.txt with info like start time, device IDs, etc.

SessionManager can return a **SessionInfo** data class containing all these file paths and metadata. For example:

data class SessionInfo(val name: String, val dir: File,   
 val rgbVideoFile: File,   
 val irVideoFile: File,  
 val rawImageDir: File?,  
 val shimmerFile: File)

This SessionInfo is passed to recorders so they know where to output. The SessionManager ensures directories exist (create with File.mkdirs()) and that existing files won’t collide (if a file exists, append a suffix or increment).

*Design note:* By centralizing this, we ensure all files from one session are grouped, which is important for data management. Using timestamps in names makes it easier to organize later[[18]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Shimmer%20sensor%20integration%3A%20Shimmer%20provides,data%20stream%20at%20start%2Fstop%20events). We also plan for *scalability* – if later we have multiple phones or sessions, consistent naming helps avoid confusion.

SessionManager will also have utility to maybe clean up old sessions or compute total storage used (for future considerations, e.g., warn if storage is low before recording). As a checkpoint, after a recording, you can manually verify that a new folder was created and contains all expected files named correctly.

### **CameraRecorder** (RGB Camera Module)

**Role:** Handles the main phone camera (RGB) using Camera2 API to record a 4K video and optionally capture RAW images.

**Initialization:** On creation, CameraRecorder will set up the CameraManager and determine the camera ID to use (likely the rear camera with highest resolution). It should check that the camera supports the needed output sizes and formats. (On some devices, 4K + RAW + preview is heavy; we assume a capable device with Level\_3 hardware level[[19]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=processing,resolution%20stills%20are).)

**Starting recording:** When startRecording(sessionInfo) is called, the CameraRecorder will: - Open the camera (CameraManager.openCamera) asynchronously. We need to provide a camera background thread or use a coroutine with suspendCancellableCoroutine for the open callback. Use a HandlerThread “CameraThread” for Camera2 callbacks, or use the new camera2 concurrency if available. - Once camera is opened (CameraDevice obtained), configure a CameraCaptureSession with multiple output targets: - **Video target:** Use MediaRecorder to capture video. Configure it for 4K resolution and appropriate bitrate. Prepare the MediaRecorder with the output file (sessionInfo.rgbVideoFile). Set video source from camera, audio source (mic) if audio included, output format (MPEG\_4), encoder (H.264 or H.265). Call MediaRecorder.prepare() before creating the capture session. - Get a Surface from MediaRecorder via recorder.surface. - **Preview/stream target:** Create a ImageReader for a lower resolution (say 1280x720) with format YUV\_420\_888 for preview frames. Its surface can be used both for local preview (if we render to a TextureView using that image) and for sending to PC. Alternatively, if using a TextureView for on-screen preview, we would have a Surface from a SurfaceTexture. However, since the UI preview can directly use the camera output (we can include the TextureView’s Surface as another output target), we might not need an ImageReader for preview. Another approach: **Camera2** can stream to a TextureView (for on-phone preview) and simultaneously to an ImageReader (for PC streaming). We have up to 3 targets on Level\_3 devices[[20]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=up%20to%203%20streams%20if,tested%20on%20the%20specific%20hardware). - **RAW target (optional):** If device supports RAW, create an ImageReader with format RAW\_SENSOR at full resolution. However, continuous RAW at 30fps is usually not possible. A strategy: set up the RAW ImageReader but don’t add it to the session initially (or if added, only capture from it when needed, not constantly). We might capture a RAW frame at start or at some interval (e.g., one RAW per second for analysis) to avoid huge overhead.

The outputs might look like:

val surfaces = mutableListOf<Surface>()  
surfaces.add(mediaRecorder.surface) // 4K video  
surfaces.add(previewSurface) // from TextureView or ImageReader for preview  
if (captureRaw) surfaces.add(rawImageReader.surface)  
cameraDevice.createCaptureSession(surfaces, stateCallback, cameraHandler)

- In the CaptureSession.onConfigured, start recording: - Start the MediaRecorder (it begins capturing frames to file). - Submit a repeating capture request to the CameraCaptureSession:

val request = camera.createCaptureRequest(CameraDevice.TEMPLATE\_RECORD).apply {  
 addTarget(mediaRecorder.surface)  
 addTarget(previewSurface)  
 // (no RAW here; RAW can be captured via separate request)  
 set(CONTROL\_MODE, CONTROL\_MODE\_AUTO)  
}  
session.setRepeatingRequest(request.build(), null, cameraHandler)

This drives the camera at the video frame rate, outputting to both video and preview. - If we want to grab RAW occasionally, we can either create a second session or use session.capture() with a TEMPLATE\_STILL\_CAPTURE targeting the RAW surface, while the repeating request is ongoing (some devices allow it).

* The CameraRecorder should also handle **focus/AE** as needed (likely set continuous focus, etc., via capture request controls).
* **During recording:** It could listen for certain events (if needed, e.g., to know when a frame is captured for timestamping, etc.). But mostly, MediaRecorder handles writing video. We might want the exact start time – we can note System.nanoTime() at the moment we call MediaRecorder.start() and use that as video start timestamp.

**Stopping recording:** On stopRecording(), the CameraRecorder will: - Signal MediaRecorder to stop (mediaRecorder.stop()), which finalizes the MP4 file. This can be a blocking call that takes a moment to write trailers. - Close the CameraCaptureSession and CameraDevice (to free them). - Release the MediaRecorder. - Close any ImageReaders (and retrieve any last frames if needed, e.g., if RAW was captured, save them).

**Preview integration:** For local on-phone preview, the easiest is to attach the camera output to a TextureView in MainActivity. We could have CameraRecorder accept a Surface or TextureView from the activity for preview. Alternatively, PreviewStreamer could feed the preview to the UI. Simpler: let MainActivity handle showing a preview (especially since our UI is minimal). We can use a TextureView and once the camera is open, give its SurfaceTexture to camera. In summary, CameraRecorder can expose a method setPreview(surface: Surface) to link the UI preview.

**Preview for PC:** The ImageReader approach allows us to get YUV frames in code. The PreviewStreamer (next module) can subscribe to these frames (set an OnImageAvailableListener on the ImageReader) and then compress/send them. We configured target 2 for preview YUV[[21]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=recorder%20,tested%20on%20the%20specific%20hardware). This way the preview frames can be consumed without disturbing the main recording (Camera2 can handle multiple outputs if the device supports it). Note: if performance issues arise (4K + preview), we may lower preview frame rate or resolution to not overload device[[22]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Handling%20high,device%20when%20doing%20multiple%20tasks).

**Concurrency considerations:** Running two cameras (RGB and IR) at once is challenging on many phones[[23]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Camera%20Capture%20,does%20support%20it%2C%20you%20can). We rely on either the phone supporting dual cameras or using an external camera for IR (our case). Since Topdon IR is external (likely via USB and not taxing the phone’s internal camera hardware), we can operate it concurrently with the internal camera[[24]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Use%20an%20external%20IR%20camera,to%20integrate%20such%20cameras). We should still monitor performance (CPU usage, thermal) given 4K encoding plus USB processing. If needed, limit frame rates (maybe 30fps for RGB, and if IR camera runs at, say, 15fps, that’s fine – it will just produce fewer frames, which can still be recorded).

### **ThermalRecorder** (Infrared Camera Module)

**Role:** Interfaces with the Topdon IR camera via the provided SDK, to capture thermal video frames and record them. The specifics depend on the SDK’s API (which appears to use UVCCamera under the hood).

**Initialization:** Likely involves obtaining a USBMonitor and UVCCamera instance. The Topdon sample IRUVC class suggests: - There’s a USBMonitor that listens for device attach (the SDK may handle permission and connection). - Once permission granted, open the UVCCamera. - Set parameters (maybe resolution, frame rate, palette for thermal image, etc.). - There’s an IFrameCallback interface to receive frames (in the sample, frames might be delivered as Y16 (16-bit temperature data) or RGBA palette).

**Starting recording:** For consistency, we want to output a video file for thermal as well. If the SDK can deliver a video stream, we may have to **manually encode** it: - Possibly the SDK provides an API to get frames continuously (e.g., via IRCamera.setFrameCallback(callback)). - We can create a MediaCodec encoder for MP4 (since MediaRecorder likely doesn’t directly support a custom source easily). Alternatively, we might simulate a virtual camera feed. Simpler: we get frames (as bitmaps or byte arrays) and encode them. - Since thermal cameras often have lower resolution (e.g., 256x192), we can encode at that resolution or upsample to a standard small video size. - **Encoding approach:** Use MediaCodec with an H.264 encoder. Initialize it with width/height equal to the frame size, color format maybe YUV420 (if we can convert the incoming frame), and configure an MP4 muxer. This is complex; a simpler (but possibly slower) path is to use an external library or record individual images. However, saving each frame as an image could lead to hundreds of images – instead, better to produce a video for synchronization. - Perhaps the Topdon SDK has a recording utility. If not, we implement as above.

**Frame handling:** The Topdon IR likely gives either: - a thermal image (grayscale or pseudo-color). Possibly in Y16 format (16-bit per pixel representing temperature). We may need to convert to an 8-bit grayscale or colorized RGB for viewing/encoding. The SDK might have a utility to get a temperature matrix or a bitmap. - We should decide on what to record: the *visual* IR image or the raw temperature data. For simplicity, record the visual image (which can be a grayscale video).

**Preview:** The IR camera frames could be shown on the phone too (the app might show both RGB and IR previews). But one phone screen might not easily display both simultaneously unless in a split view. Perhaps just switching preview or showing one in a small overlay. For now, we can skip on-phone preview for IR to focus on recording. Instead, the PC preview will show it.

**Stopping:** Stop frame capture callback, close the UVCCamera and USB resources. Finalize the video file if we were encoding.

**Threading:** The Topdon library likely uses its own threads/callbacks. We might not need to use coroutines heavily here except to offload encoding. We can set up a coroutine to consume frames from a queue and feed the encoder.

**Important:** Ensure the USB permission workflow: either rely on the manifest intent filter for permission or explicitly request via usbManager.requestPermission(device, pendIntent). The USBMonitor from the SDK probably does that if we call USBMonitor.register(). We should integrate that in ThermalRecorder.start() – e.g., call usbMonitor.register() to start listening (which will pop a system dialog for permission unless already granted), then on permission, open camera and start streaming frames.

**Sample integration:** The sample app’s manifest and code show meta-data for device\_filter.xml (with vendor/product). We should include the same XML (from ANDROID\_SDK\_USB\_IR\_1.3.7/libir\_sample/usbir/src/main/res/xml/device\_filter.xml in the repo) in our app’s res/xml. That likely covers known device IDs. This way, when our app starts, if the device is plugged, the system will grant permission (or ask once).

**Testing approach:** Initially, implement ThermalRecorder to simply get frames and perhaps log frame arrival (to ensure we get data from the device). We can postpone full video encoding if time is short – as a placeholder, we might write timestamped PNG images for a few frames or a short buffer, just to validate pipeline. But since the milestone calls for “thermal video”, we aim to produce an MP4 as deliverable.

### **ShimmerRecorder** (Bluetooth Shimmer Sensor Module)

**Role:** Connects to the Shimmer sensor via Bluetooth, starts streaming sensor data (e.g., GSR, ECG, accelerometer), and logs the data with timestamps. It ensures the data is synchronized to the recording session.

**Initialization:** Likely uses Shimmer’s API class (for example, Shimmer provides ShimmerDevice or similar). The workflow: - Scan for devices (if not pre-paired, might need a scan which requires BLUETOOTH\_SCAN). - Or if the device BT MAC is known and already bonded, directly connect by MAC using the API. - The Shimmer API might handle connecting and provide a callback or listener for incoming data packets.

**Starting recording:** On startRecording(sessionInfo), ShimmerRecorder will: - If not already connected, initiate connection to the Shimmer sensor. (In practice, we might want to connect in advance to save time, but let’s assume we connect at start for now, or ensure it’s connected by the time we start recording.) - Once connected (this could be synchronous or asynchronous), configure the sensor: - e.g., set sampling rate (like 51.2 Hz or whatever needed), - enable specific sensor channels (the API may have methods like shimmerDevice.enableSensors(Sensor.GSR, Sensor.ECG) etc., depending on what data we need). - Start streaming. The Shimmer API likely has a method startStreaming() that begins sending data packets. - Create a file (CSV or TXT) at sessionInfo.shimmerFile. Write a header line (sensor names, units). - As data arrives (the API might use a callback or an event system), write each sample to the file. Each line might include timestamp (relative or absolute), and sensor values. If multiple sensors, multiple columns. - Include a timestamp synchronization: when recording started, mark t=0 for sensor relative time, or record an absolute reference (e.g., phone SystemClock elapsed time or a Unix epoch time). We might also log a special line “START\_RECORD @ [phone\_time\_ms]” at the top of the file.

We should use a separate coroutine or thread for reading the data to not block anything. Possibly the Shimmer API callbacks are already on a background thread (to be verified). If not, we can dedicate a coroutine on Dispatchers.IO that waits for data from an input stream.

**Buffering strategy:** To avoid too frequent disk writes (and to sync with PC, if needed), we might buffer readings and flush periodically. But given the data rate of Shimmer is relatively low (e.g., even a few hundred Hz with a few channels is fine), we can write line-by-line. Still, using a buffered writer is recommended.

The plan suggests optionally forwarding sensor data live to PC for monitoring[[25]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=the%20app,with%20too%20many%20small%20packets). For now, we can log locally; but we keep in mind to integrate with SocketController later (maybe sending periodic summaries or so).

**Stopping:** On stopRecording(), signal the Shimmer to stop streaming (shimmerDevice.stopStreaming()), and close the connection (or keep it if we plan to reuse). Close the file writer so that data is saved.

Also, possibly write a footer or final timestamp (like “STOP\_RECORD @ time”). Then the file can be later merged with video timelines using the common time reference.

**Synchronization:** Shimmer’s data will have its own timestamps (some Shimmer packets include timestamp or sequence number). We should capture an initial offset – e.g., record the phone’s system time at the start of recording, and perhaps also each data packet’s device timestamp if available. This will allow aligning the sensor timeline with video timeline in post-processing. Since the PC triggers start for both phone and Shimmer, we assume all start near-simultaneously. For now, within ~100 ms accuracy is okay, and we can refine later (maybe by sending a start trigger to Shimmer or aligning via an external event like a clap visible in video and spike in sensor).

**Error handling:** If the Shimmer disconnects mid-session (BT drop or out of range), the recorder should log an error or attempt reconnection (if quick). At least ensure the file is properly closed and perhaps mark discontinuity. Those details might be beyond initial implementation, but keep it in mind.

### **SocketController** (Network Communication Module)

**Role:** Manages the socket connection between the phone and PC. It’s designed to **listen for incoming commands** (like start/stop recording) and to send status or data messages (like preview frames, or acknowledgements). In Milestone 2.1, we implement a basic scaffold: perhaps it connects to a known IP or opens a port and logs data, without full command set. But the structure is laid out for future expansion.

**Design:** We decide whether the phone is the **client** or **server**. A typical approach: PC acts as server, waiting for phone connections (because PC has a stable IP and can coordinate multiple devices). Alternatively, phone connects to PC (client) if PC runs a server – this is probably easier (PC can run a Python socket server). Let’s assume **PC is server**, phone is client.

So SocketController will: - Hold the server’s IP and port (could be configurable or hard-coded for now). - Use a background thread or coroutine (on Dispatchers.IO) to create a Socket and connect to the PC’s socket server. - Once connected, possibly send an initial hello (like sending the phone ID or status). - Then continuously listen on the socket input stream for incoming messages. - Use a simple protocol, e.g., newline-delimited text commands or a lightweight JSON.

For example, PC might send a line: "CMD START\_RECORD", or JSON {"cmd":"START\_RECORD"}. The SocketController reads it, parses it, and then invokes a callback to RecordingService (e.g., call service.beginRecordingSession() as shown earlier). Similarly, it could listen for STOP.

Also, SocketController can send messages. For instance, after starting recording, phone could send back "STATUS RECORDING\_STARTED" or even stream data (like, sending preview images or sensor data in real-time). For now, we will mainly implement receiving commands, and possibly sending simple status replies (to confirm command execution).

**Threading:** We’ll run the socket listening loop in a dedicated coroutine:

class SocketController(val serviceCallback: (String) -> Unit) {  
 private var socket: Socket? = null  
  
 fun startListening() = CoroutineScope(Dispatchers.IO).launch {  
 try {  
 socket = Socket(serverIp, serverPort)  
 val reader = BufferedReader(InputStreamReader(socket.getInputStream()))  
 val writer = PrintWriter(socket.getOutputStream(), true)  
 // notify connection success  
 Logger.log("Socket connected to PC")  
 // Listen loop  
 var line: String?  
 while (socket!!.isConnected && reader.readLine().also { line = it } != null) {  
 line?.let { processCommand(it, writer) }  
 }  
 } catch (e: Exception) {  
 Logger.log("Socket error: ${e.message}")  
 }  
 }  
  
 private fun processCommand(cmdLine: String, writer: PrintWriter) {  
 Logger.log("Received command: $cmdLine")  
 // Simple protocol parsing  
 when(cmdLine.trim().uppercase()) {  
 "START\_RECORD" -> {  
 serviceCallback("START") // signal service to start  
 writer.println("ACK STARTED")  
 }  
 "STOP\_RECORD" -> {  
 serviceCallback("STOP")  
 writer.println("ACK STOPPED")  
 }  
 // ... other commands like "PING", etc.  
 }  
 }  
  
 fun stop() {  
 socket?.close()  
 }  
}

In the above pseudo-code, serviceCallback is a lambda that the RecordingService provides, so the controller can tell the service to start or stop (the service might then call its internal methods or send an intent to itself). We also send back an "ACK" to PC for confirmation.

This is rudimentary – in a real scenario, we’d likely use a more robust message format (JSON with fields), and handle things like PC requesting a preview frame or calibration capture. But the scaffold is here.

We should also ensure the socket doesn’t block the main thread – hence using Dispatchers.IO. Also handle reconnection logic: if the PC isn’t reachable, maybe keep trying or notify the UI. Possibly include a small delay and retry mechanism. For now, if connect fails, we log and continue without socket (the local UI can still start recording).

**Security:** In a closed lab setting, an unauthenticated socket is fine. If needed, we could implement a simple auth or ensure it only connects to known IP.

**Testing:** Initially, you can test this by running a simple TCP server on your PC (e.g., using nc or a small Python script) and see that when the phone app starts, it connects. Then type "START\_RECORD" on PC side and see if the phone logs it and starts recording. This will be a major milestone test – verifying remote control. If not immediately needed, you might skip actual PC control until Milestone 2.2, but having the code path in place is valuable.

### **PreviewStreamer** (Live Preview Streaming)

**Role:** Handles sending live preview frames to the PC (and/or updating the local UI preview, depending on design). This component interacts closely with CameraRecorder (and possibly ThermalRecorder) to obtain preview frames.

For the RGB camera, as discussed, we set up an **ImageReader** for preview frames. We can give PreviewStreamer access to that ImageReader’s listener. When a new frame arrives: - In the OnImageAvailableListener, it acquires the Image, converts it to a desired format (e.g., JPEG or a downscaled JPEG), and sends it via the socket. - We likely don’t want to send full-resolution frames due to bandwidth; since we configured the preview at a lower resolution (say 720p), that’s better. Still, compress to JPEG to reduce size. Using Image.getPlanes()[0].buffer we get the YUV data, which we can convert to a JPEG. We could use ScriptIntrinsicYuvToRGB and then compress to JPEG, or easier, use the Android ImageReader in **JPEG** format directly for preview (that way the camera HAL gives us JPEG-compressed frames). However, camera2 might not allow a second output as JPEG while the main output is recording. Alternatively, *if using CameraX*, one could get an ImageProxy and use ImageProxy.toBitmap() easily – but let’s assume Camera2 low-level: - Possibly use an **additional** MediaCodec encoder to encode the preview YUV to an H264 stream and send that. That’s heavier but more efficient if streaming video. A simpler approach is sending JPEG stills periodically (like a MJPEG stream). For now, let’s plan on **periodic JPEG frames** (e.g., 1-5 fps) for preview just to verify alignment and focus remotely, not full-motion video (which could overwhelm Wi-Fi).

**Implementation:** PreviewStreamer runs in its own coroutine as well: - It may subscribe to frames via a channel or callback from CameraRecorder. For example, we can give CameraRecorder a reference to previewImageReader. Then:

previewImageReader.setOnImageAvailableListener({ reader ->  
 val image = reader.acquireLatestImage() ?: return@setOnImageAvailableListener  
 previewStreamer.onFrameAvailable(image)  
 image.close()  
}, backgroundHandler)

- The onFrameAvailable(image) in PreviewStreamer will launch a coroutine on Dispatchers.IO to handle conversion and sending:

fun onFrameAvailable(image: Image) {  
 GlobalScope.launch(Dispatchers.IO) {  
 val jpegBytes = convertYUV420ToJpeg(image) // implement conversion  
 socketController?.sendBytes(jpegBytes) // ensure SocketController has a method to send bytes  
 }  
}

The SocketController.sendBytes could simply write to the socket output stream. We might prepend some header or use a separate port for a raw stream. Possibly easier: run a separate RTSP or HTTP server for preview – but that’s out of scope for now. We can stick to a simple approach: for example, send each JPEG preceded by its length in bytes. The PC client would then reconstruct. Since this is complex to get perfect in limited time, the milestone might accept even if preview streamer is just a stub or only local. The key is designing for it.

* If streaming both RGB and IR previews, we might tag them. E.g., send a message PREVIEW\_RGB:<base64 jpeg> etc. Or use separate sockets for each feed.

**Local Preview UI:** The MainActivity likely has a SurfaceView/TextureView for local preview of at least the RGB camera. We can reuse the preview frames for that or let camera directly feed the TextureView (which is simpler and higher FPS). We could do: cameraRec.startPreview(textureView.surface) before recording. That would show a live image on phone. Since the milestone specifically mentions a *UI to trigger local test recording*, presumably they want to see preview on the phone to aim the camera. So yes, implement phone preview: - Add in MainActivity a <TextureView android:id="@+id/textureRgb" ...> (or use CameraX PreviewView if that path was chosen). - In CameraRecorder, when opening camera, if a TextureView is available and ready, add its surface to the capture session (for preview). If not, at least one of the outputs is a low-res for either preview or streamer. We may need to juggle surfaces because if we already use ImageReader for streamer, we might not also attach the TextureView (that would be 3 outputs including video which might still be okay on Level\_3 devices). If it’s problematic, an alternative is to show the preview on phone only before recording starts (just to frame) and perhaps hide it during recording if resources don’t allow concurrent preview + recording.

We should test on target device; many can handle it (some support an encoder + preview at once[[19]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=processing,resolution%20stills%20are)).

**Stopping PreviewStreamer:** When recording stops, or if preview streaming is no longer needed, we stop capturing frames. E.g., remove the OnImageAvailableListener or just ignore incoming frames. Also ensure to flush/close any streams.

### **Logger** (Logging Helper Utility)

**Role:** Provide a convenient way to log debug messages from all modules, and possibly write to a log file. We create a simple static utility (or object in Kotlin):

object Logger {  
 private const val TAG = "MultiRecorder"  
 fun d(component: String, msg: String) {  
 Log.d(TAG, "[$component] $msg")  
 // Optionally append to a file in session or global logs  
 }  
 fun e(component: String, msg: String, throwable: Throwable? = null) {  
 Log.e(TAG, "[$component] ERROR: $msg", throwable)  
 // Also append to file  
 }  
}

This way, each class can do Logger.d("CameraRecorder", "Camera opened"). During testing, we watch Logcat for [CameraRecorder] or errors. We can enhance this to write to a persistent log (maybe one per session in the session folder, or a rolling app log). But initially, logcat output and maybe toasts for critical issues (e.g., “Camera error, recording stopped”).

This Logger centralization is optional, but it ensures consistent tagging and easier enabling/disabling of verbose logging. Alternatively, as noted, using Timber.plant(DebugTree()) and then Timber.tag("CameraRecorder").d("...") is fine too.

### **FileUtils** (File Utility Helper)

**Role:** Contains common file operations, e.g., ensuring directories exist, calculating filename timestamps, etc. We might integrate this into SessionManager directly. But if more is needed: - A function to check free space: we could call StatFs on the storage path to ensure e.g. at least X MB free before starting recording (and warn if not). - A function to safely create a unique file: (though SessionManager already does by using timestamp). - Utilities to write text to a file (though can do with normal Kotlin file APIs).

We can have, for instance:

object FileUtils {  
 fun createDirIfNotExists(dir: File): Boolean {  
 if (!dir.exists()) {  
 return dir.mkdirs()  
 }  
 return true  
 }  
 fun getTimestampString(): String {  
 val sdf = SimpleDateFormat("yyyyMMdd\_HHmmss", Locale.US)  
 return sdf.format(Date())  
 }  
}

These can be used by SessionManager.

Additionally, if needed, handle MediaStore insertion if later we want to add the video to gallery (not critical for now since we can fetch via file manager or USB).

All these classes will work together as follows when the app runs:

**MainActivity** creates an intent to start RecordingService (on Start button click). The service in onCreate initializes everything (including starting SocketController if needed, and showing the “Idle” notification). When user hits “Record” on UI, we send ACTION\_START\_RECORD intent to the service (or use startService again with that action, since the service is sticky). The service then uses SessionManager to prep file paths and calls each Recorder’s start. CameraRecorder opens camera and starts MediaRecorder (the LED on phone might turn on if camera in use), ShimmerRecorder connects and streams, ThermalRecorder opens IR cam and streams. The service updates its ongoing notification (“Recording…”). Meanwhile, MainActivity can also update UI (maybe show a red dot or timer via bound service or LiveData – though a simple approach is to disable the Start button and enable a Stop button).

If the user taps Stop in UI, we send ACTION\_STOP\_RECORD. Service stops all recorders, finalizes files, and either keeps running (if we expect multiple sessions, maybe it stays active waiting for next command) or stops itself. We can keep it running to listen for socket commands indefinitely, but since this is a test recording mode, we might stop it. The UI can then show “Recording saved to ...”.

Throughout, each component logs status: - e.g., CameraRecorder logs “CameraRecorder: Video file saved at ...”. - ShimmerRecorder might log “Received 1200 samples in 60s” at end, etc. These logs and the presence of files are how we verify correctness.

## MainActivity UI for Testing

The MainActivity provides a basic UI to trigger and observe recordings on the device itself, which is vital for development and local testing (before the PC control UI is ready). The UI should include:

* **Start/Stop Button:** A toggle or two separate buttons to start and stop the RecordingService. Initially a “Start Recording” button is enabled; when clicked, it should check (and request) all needed permissions first (Camera, Audio, BT, Storage, etc.), then call ContextCompat.startForegroundService(...) with an intent for RecordingService. After starting, the button might change to “Stop Recording” (or that becomes enabled) to allow stopping. On stop, call stopService(...) or send an intent with ACTION\_STOP (as described).
* **Status Display:** A simple TextView or indicator to show status (e.g., “Idle / Recording / Saved to path”). This could be updated via local broadcast or if we bind to the service. Simpler: the service could broadcast an intent on status changes (like “com.example.MULTIRECORDER.STATUS” with extras). But given time, even updating UI optimistically is okay (Start button pressed -> show “Recording…”).
* **Preview windows:** As discussed, ideally show at least the RGB camera preview so the user can frame the shot. For this, include a TextureView or SurfaceView. For example,
* <TextureView  
   android:id="@+id/texturePreview"  
   android:layout\_width="match\_parent"  
   android:layout\_height="200dp"  
   android:layout\_centerInParent="true"/>
* (200dp as a small preview strip, or use match\_parent to cover screen if you want full preview). The IR preview could be another view below or toggled. Perhaps one TextureView that can switch between cameras if needed, but since IR is external, it might not integrate easily into the Camera2 API for preview. If the Topdon SDK provides a SynchronizedBitmap or some view class for the IR, we might use an ImageView that we update with the latest frame (converted to Bitmap). For initial test, you might skip IR preview on phone and just trust it records, due to complexity.
* **Connect/Shimmer status (optional):** Maybe a small text that says “Shimmer connected” or “Connecting…” since Bluetooth might take a second. The ShimmerRecorder can broadcast or we can check it after start.

Since the UI is mainly for testing, it can be simple (even just two buttons and a log text area). Keep it user-friendly if possible: e.g., disable Start if permissions missing, or prompt user.

**Permission requests in UI:** On app launch, we should prompt for Camera, Audio, Storage, Location/Bluetooth as needed. This can be done in MainActivity.onCreate or when Start is pressed. It’s often better to do upfront so that by the time the service runs, it has what it needs. Use ActivityCompat.requestPermissions for each group and handle the callback. You might request multiple together (Camera, Audio, Storage in one go, then Bluetooth permissions in another because they’re separate “Nearby devices” dialog). Ensure to educate user (via dialog text) why each is needed (“Camera permission is required to record video”, etc.).

**Binding to Service (optional):** We might not need to bind; sending intents might suffice. But if we wanted, we could bind to get status updates or to call methods directly. This adds complexity, so for now, Start/Stop via intents is enough.

In summary, **workflow** on UI: 1. On launch, request permissions. 2. On “Start Recording” click: - Ensure permissions granted (if not, request and return). - Show a message “Starting…” - Call startForegroundService(Intent(this, RecordingService::class.java).setAction("ACTION\_START\_RECORD")). - Maybe also disable the Start button and enable Stop. 3. The RecordingService will post its notification. The UI might not automatically know when recording actually started, unless we implement a callback. We could cheat and assume it starts near instant, and change status text to “Recording”. 4. On “Stop Recording” click: - Call startService(Intent(this, RecordingService::class.java).setAction("ACTION\_STOP\_RECORD")) (or stopService which calls onDestroy, but better to let service handle cleanup). - Show “Stopping…” then “Saved to XYZ” when done. We could catch a broadcast from service when done. Alternatively, the service could simply stop itself, and we know that once we call stop, after a short delay files are written. For testing, we might manually check the folder to verify.

You can incorporate a **Toast** or dialog at the end like “Recording saved in /Documents/MultiRecorder/Session\_xxx”.

**Layout design:** Since usability is not the main focus (it’s a research tool interface), keeping it minimal is fine. But ensure all needed info is visible for testing (like the path of files or any error messages).

## Coroutine and Threading Strategy

Using Kotlin coroutines allows us to manage asynchronous tasks more cleanly than raw threads. Here’s how we apply it in this project:

* **Main thread vs Background:** All camera and IO operations must be off the main thread to avoid freezing the UI[[2]](https://developer.android.com/develop/background-work/services#:~:text=Caution%3A%20A%20service%20runs%20in,ANR%29%20errors). We use Dispatchers.IO for file I/O (writing sensor data, socket reading/writing, encoding frames) and use Dispatchers.Default or a dedicated single-thread context for camera operations if needed (Camera2 requires a Looper thread; we might stick with a HandlerThread for camera to feed into capture session API, which is typical. We can integrate that with coroutines by using launch with a specific dispatcher or withContext around blocking calls).
* **Service CoroutineScope:** As described, RecordingService will create a CoroutineScope (with a SupervisorJob) for a recording session. This scope’s lifetime is tied to the recording task; when recording stops or service destroyed, we cancel it to stop all child coroutines (ensuring cameras and sockets shut down promptly).
* **Parallel tasks:** We identify tasks that can run in parallel:
* Capturing from two cameras concurrently.
* Writing sensor data concurrently.
* Listening to socket commands concurrently (should always be running in background to catch a “STOP” from PC, even while recording).
* Streaming previews concurrently. Coroutines (with separate dispatchers) are perfect for this, as they let us not worry about explicit thread management and synchronization (as long as each subsystem mostly operates independently or communicates via thread-safe mechanisms like channels).
* **Thread safety and Shared Data:** Avoid shared mutable state across coroutines as much as possible. For example, each Recorder encapsulates its state. If they need to report something to service, they can use thread-safe callbacks or send a message via a Handler or LiveData. The service can coordinate the results if needed. We can use synchronized or locks if ever needed (like for writing to a common log file from multiple threads, but writing to separate files per modality avoids a lot of locking needs).
* **Lifecycle awareness:** If the app goes to background, since we’re a foreground service, we continue. If the user closes the UI, the service keeps going. If the user kills the app (swipes from recents), the system might kill the service too unless it’s truly foreground – but since we are, it should remain. We should handle onDestroy in service to gracefully stop coroutines. Also, handle onTaskRemoved (if user swipes away app, call stopSelf() maybe). But in research context, they’ll likely intentionally stop via UI or PC, not kill the app mid-record.
* **Cleanup with coroutines:** Use try-finally or invokeOnCompletion on jobs for cleanup tasks. E.g., if a coroutine launching camera recording throws an exception, ensure the camera is closed in a finally block there or catch in service and stop everything. SupervisorJob allows siblings to continue if one fails; we may actually want to propagate failure – e.g., if CameraRecorder fails to start (camera error), it might make sense to abort the whole recording. In that case, we wouldn’t use Supervisor for that, or we’d manually detect and stop. For now, we can keep it simple: check return statuses from start functions and if any fail, call stop on others and report error.
* **Example:** Shimmer read loop might look like:
* scope.launch(Dispatchers.IO) {  
   shimmerDevice.startStreaming()  
   while(isActive) {  
   val packet = shimmerDevice.getPacket() // hypothetical blocking call  
   process(packet)  
   }  
  }
* Using isActive (checks if coroutine scope is still active) will naturally break out when we cancel the scope on stop.
* **UI and coroutine:** If we wanted to update UI from background, we’d use withContext(Dispatchers.Main) to e.g. set a TextView. But better, we can send a LocalBroadcast or use LiveData. For quick testing, maybe use a handler to post to UI. However, since our UI is very minimal, we might not need continuous updates from background except maybe showing a timer. A simple approach: use Handler.postDelayed in Activity to update a timer TextView every second while recording (checking a isRecording flag perhaps via service).

In summary, coroutines give us a structured way to manage these asynchronous flows, and by carefully scoping them we ensure no resource leaks (all child coroutines cancelled on stop). We also leverage the fact that many Android components (Camera, MediaRecorder callbacks, etc.) still use listeners, but we can wrap or launch coroutines from those callbacks to hand off heavy work.

## Manual & Unit Testing Checkpoints

To ensure each part works correctly and the overall system meets requirements, we plan the following **testing steps and checkpoints**:

1. **Gradle Build and Dependency Check:** *Before coding*, verify the project compiles with all dependencies. For example, include a small snippet using Shimmer API (like referencing a Shimmer class) to ensure the library is resolved. If the Topdon .aar is included, ensure it’s picked up (no NoClassDef when running). **Checkpoint:** Build succeeds, app launches on phone (even if it does nothing yet).
2. **Permission Flow Test:** Run the app on a device and ensure that upon pressing Start, it requests all needed permissions. Grant them and ensure no crashes. **Checkpoint:** All permissions (Camera, Audio, Storage, Bluetooth, Location for BLE) can be granted and the app handles denial (e.g., if Camera denied, show a message and don’t proceed).
3. **CameraRecorder Test (Standalone):** Write a small routine (perhaps triggered by a debug button in MainActivity) that uses CameraRecorder to record a 5-second video to a test file. This is to verify camera and MediaRecorder setup on the device:
4. Check that the video file is saved, playable, and is 4K resolution with expected length.
5. If RAW capture is implemented, check if RAW images are saved (and viewable by a RAW viewer).
6. Also verify that if preview is shown, it’s working (the preview TextureView displays the camera feed).
7. We might initially test with a lower resolution to simplify, then ramp up to 4K once basic works, because 4K can sometimes reveal performance issues. **Checkpoint:** RGB video recording works: file exists in session folder, correct resolution, not corrupted (play it back). No crashes when starting/stopping repeatedly.
8. **ThermalRecorder Test:** This requires having the Topdon IR device connected. Steps:
9. Launch the app with device connected (or connect it after launching). Check if the USB permission dialog appears and is handled (our manifest filter should catch it).
10. Call ThermalRecorder.start() in isolation (maybe from a button). Possibly have it run for a few seconds and then stop.
11. Because encoding to video is complex, for the first test, simply try to retrieve a frame: e.g., after start, after 1 second, grab one frame via the SDK and save as PNG to storage, then call stop. This will confirm we can communicate with the camera.
12. Once basic frame retrieval works, test continuous capture: e.g., capture frames for 5 seconds and count them. Then try integrating the video encoding pipeline.
13. Check the output IR video file (or sequence). Ensure the frames look correct (perhaps compare with the Topdon sample app output if available). **Checkpoint:** IR camera connects and yields data. Ideally, an IR video file is saved and can be played (e.g., resulting MP4 shows the thermal imagery changing over time). If MP4 is tricky, at least a series of images or a short raw binary dump that we can interpret means success. We also check that running the RGB and IR recording together doesn’t crash (the ultimate test will be with both on).
14. **ShimmerRecorder Test:** Without involving camera, test connecting to Shimmer:
15. Ensure the Shimmer is powered and in range. Possibly pair it via Bluetooth settings first (if required by Shimmer device).
16. Have a button to connect Shimmer (or automatically on Start). Monitor logs to see if connection succeeds (the Shimmer API might log to Logcat, or we add Logger lines in callbacks).
17. If available, use a Shimmer emulator or read from a recorded file in absence of hardware (Shimmer provides a demo mode? If not, need the actual device).
18. Once connected, start streaming for few seconds, then stop. Open the output .csv file and verify data entries (e.g., time and sensor values). If you have a known stimulus (like shake the sensor or touch GSR electrodes), see if values change accordingly in the file.
19. Check timestamps in the file to ensure they increment reasonably (no large gaps or resets). **Checkpoint:** Sensor data file is recorded with plausible values. No disconnects or, if there are, they are handled (reconnect or at least logged).
20. **Integrated Full Recording Test:** Now, test the entire pipeline together using the MainActivity UI:
21. Press “Start Recording” on the UI. The expectation:
    * Foreground notification appears (e.g., Android status bar shows our app is recording).
    * The UI preview shows the RGB camera feed.
    * The Shimmer device’s indicator (if any) shows it’s transmitting (some Shimmers have LEDs).
    * Let it run for, say, 10 seconds, then press “Stop Recording”.
22. After stopping:
    * Notification should go away (or change to something like “Recording complete” and then removable).
    * Check the session folder on storage. It should contain:
    * rgb\_video.mp4 (verify filesize > 0 and playable).
    * ir\_video.mp4 (or images if that’s how we did it).
    * shimmer\_data.csv (open and see data).
    * Any raw images or other files if applicable.
    * If we planned a metadata file, check that too.
    * Check synchronization roughly: If possible, do a **sync test**: e.g., clap once during recording in view of both cameras and maybe a sudden movement detectable by accelerometer. Later, see if the clap frame in RGB video corresponds to a bright frame in IR video (if visible) and a spike in accelerometer data at the same timestamp. This can give an idea of sync offset. It won’t be perfect, but we aim for them to start at nearly the same time. We did not explicitly sync beyond simultaneous start, so some small offset (tens of ms) might exist. That’s usually fine[[4]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=High,A%20compromise)[[26]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=might%20be%20to%20record%20the,by%20timestamping).
23. Verify that if any component fails (e.g., cover the camera lens to cause an autofocus issue, or turn off Shimmer mid-run), the app handles it gracefully (perhaps logs error but continues others, or stops everything cleanly). **Checkpoint:** Full system recording works and data from all modalities are present. This effectively delivers Milestone 2.1 functionality locally.
24. **Socket Command Test:** (If implemented now) Run a simple socket server on PC and attempt to start/stop via network:
25. On PC, use a script to listen on port (for example, Python: socket.listen()).
26. On phone, maybe add an input field in UI to enter PC IP (or hardcode if on same Wi-Fi). Ensure phone is on same network.
27. Launch service, ensure SocketController connects (Logger should show “Socket connected”).
28. From PC, send text “START\_RECORD”. See if phone responds by starting recording (look at phone, it should show notification and actually record).
29. After a few seconds, send “STOP\_RECORD”. Check that phone stopped. Also see PC side if it received “ACK STOPPED”.
30. This demonstrates remote control capability. **Checkpoint:** Phone can be remote-started and stopped via socket. (Even if this is not fully utilized until later, having it verified ensures our architecture is correct.)
31. **Code Review & Unit Tests:** Write unit tests for non-Android logic where possible:
32. SessionManager’s file naming: simulate two sessions and ensure names differ and are correctly formatted.
33. A simple test for FileUtils.getTimestampString format.
34. If any data parsing or timestamp math is done (e.g., converting Shimmer ticks to ms), test that logic with known inputs. These are small but help catch mistakes.
35. **Performance/Stability soak test:** Do a longer recording (if possible, say 5 minutes 4K) to ensure no memory leaks or crashes:
36. Check memory usage doesn’t grow unbounded (ImageReaders can leak if not releasing images, etc.).
37. Check that file sizes match expectations (~Megabytes per minute for given bitrate).
38. After stop, ensure all threads ended (no leftover high CPU usage). **Checkpoint:** The app remains stable under extended use, and all resources (camera, etc.) are released on stop (able to start a new recording again without restarting app).
39. **Extensibility check:** Consider future requirements and ensure our choices won’t block them:
    * E.g., if later we need to integrate calibration capture, can we reuse components? (Yes, we can send a command via socket to capture a frame – CameraRecorder can have a method to capture still image and save to file).
    * If multiple sessions sequentially, does SessionManager avoid overwriting? (Yes, new folder each time).
    * If adding a second phone, the architecture on each phone is similar, and PC coordinates them – that’s beyond this app, but our modular approach on one phone is a good template for multiple.

By following this implementation guide and verifying each part step-by-step, we ensure that the Android app is robust and ready for integration with the PC control system in subsequent milestones. This design emphasizes **long-term modularity**, so new features (calibration routines, additional sensors, different network protocols) can be added with minimal changes to existing code. Each component can largely function and be tested independently, which reduces integration bugs. The result is a cohesive system where an experimenter can reliably capture synchronized multi-modal data with confidence in the underlying software.

[[1]](https://developer.android.com/develop/background-work/services#:~:text=A%20foreground%20service%20performs%20some,isn%27t%20interacting%20with%20the%20app) [[2]](https://developer.android.com/develop/background-work/services#:~:text=Caution%3A%20A%20service%20runs%20in,ANR%29%20errors) Services overview  |  Background work  |  Android Developers

<https://developer.android.com/develop/background-work/services>

[[3]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Camera%20Capture%20,RAW%20images%20at%20intervals%20during) [[4]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=High,A%20compromise) [[5]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=standard,tested%20on%20the%20specific%20hardware) [[6]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Shimmer%20Integration%3A%20Utilize%20the%20Shimmer,that%20we%20want%20minimal%20delay) [[18]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Shimmer%20sensor%20integration%3A%20Shimmer%20provides,data%20stream%20at%20start%2Fstop%20events) [[19]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=processing,resolution%20stills%20are) [[20]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=up%20to%203%20streams%20if,tested%20on%20the%20specific%20hardware) [[21]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=recorder%20,tested%20on%20the%20specific%20hardware) [[22]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Handling%20high,device%20when%20doing%20multiple%20tasks) [[23]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Camera%20Capture%20,does%20support%20it%2C%20you%20can) [[24]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Use%20an%20external%20IR%20camera,to%20integrate%20such%20cameras) [[25]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=the%20app,with%20too%20many%20small%20packets) [[26]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=might%20be%20to%20record%20the,by%20timestamping) Updated\_Plan\_for\_Multi\_Sensor\_Recording\_System\_Android\_+\_PC.docx

<file://file-9JgS9hNU2GwaXbC4UsQQGa>

[[7]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/build.gradle#L18-L26) [[8]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/build.gradle#L44-L51) build.gradle

<https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/build.gradle>

[[9]](https://developer.android.com/about/versions/14/changes/fgs-types-required#:~:text=FOREGROUND_SERVICE_TYPE_CAMERA%20Runtime%20prerequisites) [[10]](https://developer.android.com/about/versions/14/changes/fgs-types-required#:~:text=match%20at%20L755%20FOREGROUND_SERVICE_TYPE_MICROPHONE%20Runtime,prerequisites) Foreground service types are required  |  Android Developers

<https://developer.android.com/about/versions/14/changes/fgs-types-required>

[[11]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/AndroidManifest.xml#L28-L36) [[12]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/AndroidManifest.xml#L10-L18) [[16]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/AndroidManifest.xml#L80-L84) AndroidManifest.xml

<https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/AndroidManifest.xml>

[[13]](https://developer.android.com/develop/connectivity/bluetooth/bt-permissions#:~:text=If%20your%20app%20targets%20Android%C2%A012,in%20your%20app%27s%20manifest%20file) [[14]](https://developer.android.com/develop/connectivity/bluetooth/bt-permissions#:~:text=%3C%21,permission%20android%3Aname%3D%22android.permission.BLUETOOTH_ADVERTISE%22) [[15]](https://developer.android.com/develop/connectivity/bluetooth/bt-permissions#:~:text=The%20,as%20shown%20in%20figure%201) Bluetooth permissions  |  Connectivity  |  Android Developers

<https://developer.android.com/develop/connectivity/bluetooth/bt-permissions>

[[17]](https://stackoverflow.com/questions/52382710/permission-denial-startforeground-requires-android-permission-foreground-servic#:~:text=Permission%20Denial%3A%20startForeground%20requires%20,so%20the%20system%20automatically) Permission Denial: startForeground requires ... - Stack Overflow

<https://stackoverflow.com/questions/52382710/permission-denial-startforeground-requires-android-permission-foreground-servic>