# Implementation Guide: ThermalRecorder Module (Milestone 2.3)

## 1. ThermalRecorder Class Structure and Interface

Design the **ThermalRecorder** as a dedicated class responsible for managing the thermal camera capture, separate from other recorders (RGB video, Shimmer, etc.) for modularity. It should expose a clear interface for the lifecycle of thermal recording: e.g. initialize(), startRecording(SessionInfo), stopRecording(), and possibly startPreview() and stopPreview() for the live feed. Internally, the class will hold references to the USB camera interface (from the Topdon SDK) and manage threads for data handling. Key fields might include:

* **USB Camera Manager** – An object from the Topdon SDK (e.g. an IRUVC or similar) that handles USB connection and frame streaming.
* **Buffers for Frame Data** – Byte arrays or buffers for the incoming **thermal image frame** and **radiometric temperature data**. For example, two buffers of size width\*height\*2 bytes (since each pixel’s temperature is 16-bit)[[1]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L60-L63). These are reused for each frame to avoid reallocation.
* **Preview Display Component** – A reference to a UI element (or a callback) where frames will be rendered for live preview (e.g. a SurfaceView or custom CameraView).
* **Networking/Sockets** – A reference to the existing socket or network client to stream preview frames to the PC controller.
* **File Output** – Handles (streams or writers) for saving raw thermal data to storage (using the session information for file naming).

The class should implement any common recorder interface if one exists (for example, if there’s an IRecorder interface for start/stop, use it). This lets the main app trigger all recorders in sync easily. It also promotes extensibility – if support for another thermal camera or data source is added later, one could create another implementation of the interface without changing core logic.

**Internal Workflow:** On startRecording(), ThermalRecorder should ensure the thermal camera is connected and configured, then begin capturing frames (both for preview and saving). It will likely spawn background tasks (or coroutines) for different duties (discussed below). On stopRecording(), it stops frame capture, closes files, and releases the camera. The **SessionInfo** (containing session ID or directory) will be used to determine the output file path and naming for the thermal data file, ensuring all modalities share a consistent session naming scheme. For example, if SessionInfo provides a folder or base name, ThermalRecorder can create a file like session123\_thermal.raw in that folder.

## 2. USB Permission Flow and Topdon SDK Integration

**USB Host Permissions:** Because the Topdon TC001/Plus is a USB device, the Android app must handle USB runtime permission. In the app’s manifest, declare USB host mode support (<uses-feature android:name="android.hardware.usb.host" />) and an intent filter for the device if desired. Typically, the Topdon SDK (Infisense library) uses a helper class like USBMonitor to abstract permission handling. On device connection, the app should request user permission via UsbManager. In the Topdon sample, the USBMonitor.OnDeviceConnectListener.onAttach() callback is used to automatically call requestPermission() when a matching device is attached[[2]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L19-L27). We can adopt a similar pattern: register a USBMonitor (or broadcast receiver if not using their class) that filters for the Topdon camera’s USB Vendor/Product ID and requests permission when detected. Once the user grants permission (or if it was already granted), the camera can be opened for streaming.

**Topdon SDK Integration:** Include the Topdon SDK libraries in the project. The provided Topdon SDK (Infisense IRUVC API) contains native code (.so) and Java classes for camera control, frame processing, and command (e.g., calibration) handling. According to the SDK docs, certain support classes (like Usbcontorl and Usbjni) must be copied without modification[[3]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L196-L203) – ensure these are added to the project as instructed. After adding the SDK, initialize the camera interface. For example, create an instance of IRUVC (or the appropriate class) with the desired mode: the SDK supports either **image-only, temperature-only, or image+temperature** streaming modes. For radiometric data, we will use the **image + temperature dual output mode**, which provides both the IR image and per-pixel temperature data in one stream. In the Topdon sample, this was done by specifying resolution height as double (e.g. 256x384) and using a dataFlowMode like CommonParams.DataFlowMode.IMAGE\_AND\_TEMP\_OUTPUT[[4]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L110-L119)[[5]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L294-L303). The SDK then returns frames that contain both image and temperature bytes.

**Device Compatibility:** Both TC001 and TC001 Plus should be supported. The SDK uses device **Product IDs** to identify compatible cameras. We should include all relevant PIDs in the USB permission filter or checks. For instance, the sample whitelist includes 0x3901, 0x5840, 0x5830, 0x5838[[6]](https://github.com/CoderCaiSL/IRCamera/blob/806fbb62ffbfab3418b82d4204bbc0efbbcc68d4/libir-demo/src/main/java/com/infisense/usbir/camera/IRUVCTC.java#L61-L69) – these likely correspond to different models (the TC001 series and others). Ensuring our USBMonitor or permission logic recognizes these IDs will allow both the original and Plus models to connect. The TC001 Plus also has a visible-light camera for “fusion”, but the IR SDK appears to treat it similarly for IR output. We will focus on the IR stream; the Plus’s visible camera feed is not accessed via the IR SDK (and the device does not stream the fused image over USB, according to spec). Our design remains extensible if future devices add channels – e.g., we could extend ThermalRecorder to handle an additional visible feed if needed, but for now we concentrate on thermal data.

**USB Connection Sequence:** Upon permission grant, open the camera through the SDK. The IRUVC class (or its variant) provides a method openUVCCamera(UsbControlBlock, ...) which we call with the permission’s control block handle[[7]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L66-L74). Next, initialize the IR processing module: e.g., create the IRCMD instance via ConcreteIRCMDBuilder with the appropriate camera type. The sample uses IRCMDType.USB\_IR\_256\_384 for a 256x192 sensor[[8]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L440-L449). This call loads calibration data and prepares for temperature calculations. We should check the returned result code and handle errors (e.g., if initialization fails, inform the user to reconnect or that the device isn’t supported[[9]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L444-L453)). After that, start the camera preview via the SDK (startPreview() or similar), which will begin streaming frames into a callback.

## 3. Frame Acquisition and Radiometric Data Buffering

Once the camera is running, the SDK will provide frames via a frame callback (often on a background thread). In our ThermalRecorder, we implement the SDK’s **frame callback interface** (e.g. IFrameCallback.onFrame(byte[] frame)). Each frame delivered in **image+temperature mode** contains two parts: the IR image and the temperature matrix. Specifically, for a 256×192 resolution device, the frame byte array length is about 256*192*4 bytes when both image and temp are included (since each part is 256*192*2 bytes)[[10]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L291-L299). The first half typically contains the *thermal image* (often in YUV422 or similar 16-bit format for IR intensities), and the second half contains the *per-pixel temperature data* (each pixel’s temperature as a 16-bit value). The SDK example confirms this splitting: it checks if (length >= imageOrTempDataLength\*2) to detect dual-mode frames, then copies the second half of the data into the temperatureSrc buffer[[11]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L291-L300)[[12]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L301-L309). We will follow the same approach: maintain two byte arrays, one for image data and one for temperature data. On each onFrame call, copy out the respective portions into our buffers. (The copy ensures we own the data outside the SDK’s memory, and we can then process it without blocking new frames.)

**Radiometric Buffer Structure:** Each pixel’s temperature is encoded typically as a 16-bit **raw value**. The actual conversion to a physical temperature (°C) might require applying calibration formulas. However, the Topdon SDK’s IRCMD module likely does that internally such that the values in temperatureSrc are already “radiometric” (possibly scaled in hundredths of a degree or some fixed-point Kelvin units). In practice, we should verify this by simple tests (e.g., measuring a known temperature and checking the buffer values). For now, we will treat the temperatureSrc array as containing meaningful temperature information per pixel. We can store it directly or convert if needed (see Section 6).

The **frame rate** of the TC001 cameras is ~25 Hz[[13]](https://www.topdon.us/products/tc001-plus?srsltid=AfmBOoo8fJ1BiHiNXsNYkOI6c468tmrv4hOuPYKnV6UDjQrLnYg1rCOQ#:~:text=0), so onFrame will be called up to ~25 times per second. We must handle data quickly or buffer it. To avoid dropping frames, the callback should do minimal work – ideally just copy the data to our own queue or buffers and signal another thread to handle processing (display, file I/O, etc.). The ThermalRecorder can use a producer-consumer model: the SDK callback thread produces frames (copying into a buffer ring), and worker threads consume them for saving and streaming. This decoupling is important to maintain throughput.

Additionally, to support **both 256×192 and any higher resolution** (if a future device offers it), the ThermalRecorder should not hard-code dimensions. It can query the camera’s supported sizes via the SDK. For example, after opening, the SDK provides a list of supported frame sizes (previewList); logging those can reveal if a device is 256x192 only or others[[14]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L432-L440). The code can select the appropriate mode. Since TC001 Plus uses the same IR resolution (256×192)[[15]](https://www.topdon.us/products/tc001-plus?srsltid=AfmBOoo8fJ1BiHiNXsNYkOI6c468tmrv4hOuPYKnV6UDjQrLnYg1rCOQ#:~:text=256x192)[[16]](https://www.topdon.us/products/tc001-plus?srsltid=AfmBOoo8fJ1BiHiNXsNYkOI6c468tmrv4hOuPYKnV6UDjQrLnYg1rCOQ#:~:text=Sporting%20an%20ultra,display%2C%20making%20it%20easier%20to), we won’t need a different buffer size for it, but the detection logic makes the module extensible.

## 4. Live Preview Rendering Pipeline on Android

The app should display a **real-time thermal video preview** on the phone’s UI, so users can frame the shot and verify the thermal camera is capturing correctly. We can implement this with a dedicated **Custom View** or use existing components: the Topdon SDK sample provides CameraView and TemperatureView UI components[[17]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L154-L163)[[18]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L164-L171). These likely overlay the thermal image and allow annotations (like selecting points for temperature). For our use, a simpler approach is to use a single view to show the thermal image.

**Image Format Conversion:** The data from the camera’s image buffer is in YUV422 (or a similar Y16 format). We must convert it to a displayable format (e.g., ARGB8888) for rendering. The SDK offers utility functions in LibIRProcess to do this efficiently (e.g., convertYuyvMapToARGB() for pseudocolor or converting YUV to ARGB)[[19]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L120-L128). We will utilize these native methods to convert the raw image bytes to an ARGB pixel array (or Android Bitmap). This conversion should be done off the UI thread, then the ready-to-draw image can be pushed to the UI.

**Rendering on SurfaceView/ImageView:** One strategy is to use a SurfaceView with a **Canvas**. We can spawn a dedicated **render thread** (similar to what the sample’s ImageThread does) that waits for the latest frame, converts it to ARGB, and then locks the Canvas to draw the bitmap. The conversion includes any needed rotation correction – e.g., if the camera is physically rotated 90°, we apply LibIRProcess.rotateRight90() on the image or temperature data as needed[[20]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L295-L303) to orient it upright. After conversion, create or reuse a Bitmap of the appropriate size. Because the thermal resolution is lower than the phone screen, we should scale the image for a better view. We can use Canvas.drawBitmap() with a destination rectangle that matches the view size (maintaining aspect ratio to avoid stretching)[[21]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L170-L178). This approach is fast enough given the frame size. For instance, scaling a 256×192 image to full-screen and drawing at ~15-25 fps is well within the capability of modern phones.

Alternatively, one could use an ImageView and update its bitmap each frame, but continuously calling ImageView.setImageBitmap at high frequency can cause UI thread overhead. A SurfaceView or custom drawing view is preferable for smoother updates. The sample’s CameraView uses a scaling strategy where it creates a scaled bitmap for the canvas[[22]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L174-L181); we can improve on that by using canvas drawing with a matrix or specifying dest rect, to avoid creating a new bitmap every time (thus reducing GC pressure). Essentially, allocate one Bitmap (e.g., ARGB\_8888 of 256×192) and one for the scaled output if needed, and reuse them each frame with copyPixelsFromBuffer[[23]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L144-L151) or similar method to update content.

**UI Thread Coordination:** The rendering thread can either post the final bitmap to the UI thread or directly draw on a SurfaceView’s canvas (since SurfaceView’s canvas drawing can occur on a background thread). We just must ensure thread-safety when accessing the image data buffer. Using the SynchronizedBitmap provided by the SDK (which seems to wrap a bitmap with a lock) can be helpful[[24]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L20-L23). We will protect any shared data (like the latest frame buffer) with synchronization or atomic swaps. The preview should run continuously during recording (and even when idle if we want a preview outside of recording). We might start preview as soon as the camera is connected and keep it on until the camera is disconnected or the user turns it off.

## 5. Preview Frame Compression and Streaming to PC

In parallel with the local preview, the ThermalRecorder should stream a live preview feed to the PC application over the existing socket connection. This allows the experimenter to monitor the thermal video remotely[[25]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Preview%20Monitoring%3A%20Receive%20live%20previews,to%20switch%20between%20IR%2FRGB%20preview). However, to conserve bandwidth and avoid overloading the phone, we **compress the frames** before sending[[26]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=sending%20full%204K%20raw%20frames,frame%20rate%20preview%20just%20for). The thermal frames are relatively small (the raw 256×192 image is ~98 KB, and even the Plus’s fused image is similar resolution), but uncompressed streaming at 25 fps (~2.5 MB/s) could strain the Wi-Fi network or the phone’s CPU. We will implement a strategy to send a *throttled, compressed* preview: for example, send at most ~10 fps and use JPEG compression.

**Compression**: After obtaining an ARGB bitmap for a frame (as used for the preview), we compress it to JPEG (or PNG). JPEG is preferred for continuous video because it’s much smaller and faster; slight lossy compression is acceptable for preview purposes. We can use Android’s Bitmap.compress(Bitmap.CompressFormat.JPEG, quality, OutputStream) to get a JPEG byte array. At 256×192, a JPEG can be on the order of only a few KB. We will choose a quality that balances clarity with size (perhaps ~70-80% quality is sufficient). If performance is an issue, we could compress the grayscale image rather than the colorized one, or even send the raw grayscale frame and let the PC colorize it – but that adds complexity on PC side. Given the small size, JPEG compression on the phone is fine.

**Streaming Protocol**: The ThermalRecorder should integrate with the existing socket protocol used by the system. Likely, the phones communicate with the PC via a TCP socket or WebSocket with custom messages[[27]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Network%20Sockets%20or%20HTTP%3A%20For,frames%20as%20they%20become%20available). We can extend that protocol to carry thermal frames. For example, define a message type "THERMAL\_FRAME" where the phone sends a short header (including perhaps a timestamp and frame size) followed by the JPEG bytes. The PC, upon receiving, decodes the JPEG and displays the frame in its UI (in the designated thermal preview window). This is analogous to how the RGB preview might already be handled, albeit the RGB preview might be using a different mechanism (possibly the PC could be receiving an RTSP stream or JPEG stills). If an **existing preview streaming** mechanism is in place (for instance, if the RGB camera preview is sent as JPEG frames over the socket), we should reuse it: e.g., if the phone already has a background thread sending JPEGs for RGB, we can add thermal images to that stream (tagged with an identifier for IR vs RGB).

**Frame Rate Throttling**: We deliberately send fewer frames if needed. The PC doesn’t require a full 25 fps from both phones for monitoring – even ~5-10 fps is sufficient to observe alignment[[28]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Data%20Upload%2FStreaming%3A%20While%20recording%2C%20the,adb%20pull%20via%20a%20script). We can skip some frames for streaming. For instance, maintain a counter and only compress/send every Nth frame (depending on network conditions). This reduces bandwidth and CPU usage (since compression is done less often), and prevents backlog on the socket. The **data flow** should be tuned so that streaming the thermal preview does **not interfere with the primary 4K recording or the local preview**[[29]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=sending%20full%204K%20raw%20frames,the%20operator%20sees%20a%20near)[[30]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=preview%20frames%20,adb%20pull%20via%20a%20script). In practice, JPEG compression of a small image and sending ~100 KB over Wi-Fi periodically is minor, but we will still ensure that if the system is taxed, we prioritize local tasks (recording to disk) over preview streaming. For example, if frames start queuing up (socket not sending fast enough), we drop the oldest unsent thermal frame (better to drop frames than to stall and potentially affect recording).

**Synchronization**: Include timestamps with each frame (e.g., the phone’s current time or a frame index) in the streamed data. This isn’t strictly necessary for just viewing, but could be useful for debugging latency or aligning with other data on the PC. Since our preview frames are not used for analysis, precise sync is not critical, but a timestamp can help match a thermal preview frame with roughly the same time in the RGB video if needed.

## 6. Radiometric Raw Frame File Format

A core requirement is to **save raw per-pixel temperature data** for each frame to disk for post-analysis. We need to choose a file format that preserves the full radiometric information of each pixel. Options include saving as a sequence of CSV files, a binary file, or a structured format like HDF5. We will consider trade-offs:

* **CSV (Comma-Separated Values)**: This would store human-readable text of temperatures for each pixel. For instance, each frame could be one CSV file with 256×192 numbers. While CSV is easy to open and inspect (e.g., in Excel or MATLAB), it is extremely bulky and slow for large numbers of frames. It’s noted in thermal imaging tools that exporting an image as CSV does capture all pixel temperatures[[31]](https://www.researchgate.net/post/How-can-I-get-the-temperature-at-each-pixel-of-the-IR-image-taken-by-FLIR-T250-camera#:~:text=Hi%2C), but those are typically single frames or a few images, not a video stream. Writing dozens of CSV per second will likely become I/O-bound and consume a lot of space. CSV is therefore **not ideal for real-time recording**. We might allow an *optional* CSV export of a single frame for verification purposes, but not for continuous recording.
* **Binary Raw**: A custom binary format can efficiently store the data. For example, we can create a .raw or .bin file where we write a header (with metadata like frame width, height, and possibly total frame count or frame rate) followed by frame records. Each frame record could consist of a timestamp (e.g., 64-bit microsecond timestamp) and the array of temperature values (each value 16 bits). This format would be compact and fast to write (just a file stream write per frame). However, it’s not immediately readable without a custom parser. For analysis, one would write a script (in Python, MATLAB, etc.) to read the binary file, given the known width/height and format. This is feasible and common in research settings.
* **HDF5 or Similar**: HDF5 is a scientific data format that can store datasets (matrices) with compression. For example, we could create an HDF5 file with a dataset of dimensions (num\_frames × height × width) for temperature. This is elegant for later analysis (many tools can read HDF5), and it can compress the data significantly (since thermal data often has spatial correlation). The downside is that writing to HDF5 on Android may require adding a specialized library (since Android doesn’t have native HDF5 support). There are Java HDF5 libraries, but they might add complexity and overhead. Given the time constraints and the fact that we need streaming writes, a simpler binary format might be more practical. We can later provide a conversion tool (e.g., a Python script to convert our binary dump to HDF5 or CSV offline).

**Chosen Approach:** We will implement binary recording of frames, as it offers the best performance. We will define a simple format: e.g., a 16-byte header containing an identifier and the image dimensions, followed by repeating records of [timestamp (8 bytes)][frame data (width\*height\*2 bytes)] for each frame. Each pixel’s temperature value will be stored as a 16-bit little-endian integer representing the radiometric reading. The actual unit of these values should be documented (for instance, it might be that the value = temperature in Kelvin \* 64, based on some SDK scale factor, or maybe an absolute count – we will document how to convert it). For reference, other radiometric cameras also record raw temperature data that can be exported to CSV or binary; for example, Optris’s PIX Connect software records *raw, uncompressed temperature data* and allows export to CSV or DAT (binary) for post-processing[[32]](https://optris.com/us/software/pixconnect/#:~:text=PIX%20Connect%3A%20License,Additionally%2C%20images%20and). This validates our approach of recording raw data for flexibility in analysis.

We’ll name the file something like **Thermal\_{sessionTimestamp}.dat** or integrate with SessionInfo (e.g., if SessionInfo has a session name or number, include that). If SessionInfo provides a directory path for the session, we ensure the file is created there so that later all data from a session is collocated.

**Radiometric Accuracy Consideration:** The saved values are essentially what the camera sensor + SDK provided. If needed, we will also save a small **metadata file** (or metadata block in the header) with calibration details – e.g., emissivity setting (if adjustable), and any constants needed to interpret the raw values. The Topdon device likely assumes a standard emissivity (like 0.95) and outputs temperature assuming that. If we have access to the calibration data (the SDK had arrays like nuc\_table, gain mode, etc.), we might not need to store them for end users, but it’s good to record the basics: resolution, frame rate, and maybe the fact that values are in centi-degrees Celsius or similar.

**File Size:** Recording every frame’s full data will produce large files, but manageable. Each frame is ~98 KB (for 256×192, 2 bytes each pixel). At 25 fps, that’s ~2.45 MB/s. A one-minute recording yields ~147 MB of thermal data; a 10-minute recording ~1.47 GB. On a Galaxy S21/S22 with plenty of storage, this is acceptable for short sessions, but the user should be mindful of storage if doing very long recordings. We should document this and possibly allow an option to reduce thermal frame rate or enable compression. If space becomes a concern, we could *optionally* compress the raw data with a fast lossless algorithm or simply allow the user to turn off raw saving if not needed. For now, meeting the requirement means always saving raw frames for research fidelity.

## 7. Concurrency: Threading and Coroutine Model

Managing preview, file I/O, and streaming concurrently is critical to ensure one doesn’t block the others. We will utilize multiple threads or Kotlin coroutines to partition the work:

* **Camera Callback Thread:** The USB SDK likely runs on its own thread (the USBMonitor uses a thread for callbacks[[33]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L130-L138)). This thread will invoke our frame callback with new data. In that callback, we do a minimal operation: copy the data to our local buffers (or queue) and notify other workers. We must not do heavy processing (like encoding or file writes) directly in this callback, as it could stall the USB processing and cause frame drops.
* **File Writer Thread:** A dedicated thread (or coroutine on Dispatchers.IO) will handle writing the radiometric data to disk. For example, we can use a single-thread Executor or HandlerThread for file I/O. When a frame is ready, we package it (timestamp + data) and send it to this thread (via a thread-safe queue). The writer thread pops frames in order and writes them to the file stream. This ensures disk I/O (which may occasionally block on filesystem latency) never interferes with the capture loop. It also serializes file access so that writes happen one at a time (no concurrent writes to the same file, avoiding corruption).
* **Compression/Network Thread:** We can have another background thread for compressing and sending frames to the PC. However, since we plan to throttle this, we might not need a full-time thread running at 25fps. One approach: whenever a new frame arrives, if it’s one we intend to stream (e.g., every 2nd or 3rd frame), we dispatch a task to a threadpool (or coroutine) to handle it. That task will take the latest image, compress to JPEG, and send via socket. Because network I/O and JPEG compression can both be done in background, this should not touch the main thread. Using a small pool (size 1 or 2) is wise to avoid too many concurrent compressions if frames come faster than network sends – we might simply drop frames if the previous one is still being sent.
* **UI Thread:** The main thread will be involved only in updating the preview display. If using a SurfaceView with a dedicated render thread, the UI thread might only be needed if we use an invalidate() on a custom view. We can also use a small Handler posting to Looper.getMainLooper() to update an ImageView bitmap if we choose that route. The key is to not block the UI thread; all heavy lifting should be done already when we render. Usually, we’d just do surfaceHolder.lockCanvas() in our render thread, draw, and unlockCanvasAndPost() – this doesn’t block the UI thread at all.

Using **Kotlin coroutines**, this setup can be expressed neatly: for example, use CoroutineScope(Dispatchers.IO) for file writes and network sends (they are I/O bound), and perhaps Dispatchers.Default for image processing (if CPU heavy, though here it’s minor). A Mutex or synchronized block can protect the shared frame data. If the codebase is in Java, traditional threads/handlers will be used similarly.

**Synchronization & Frame Coordination:** We should consider that a frame consists of two parts (image and temperature). We will likely handle them together for file saving (both parts go to file) and for preview (image part for display). Our callback already separates them, so it can provide one copy of the image bytes to the preview thread and one copy of the temp bytes to the file thread. Alternatively, we might not need to copy the image bytes for file saving at all – only the temperature bytes are truly needed for raw data recording (the image can be regenerated from those if needed, but it’s not required since we’re saving temperature which is richer). So, to optimize, the file writer could write just the temp array per frame (since that’s what researchers will use for analysis). We will include timestamps, which can be taken from a consistent clock (Android’s SystemClock.elapsedRealtimeNanos() or similar) at frame arrival, to each record[[34]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=synchronize%20the%20two,aligns%20with%20the%20PC%E2%80%99s%20timeline). The **SessionInfo** or an external sync mechanism might also provide a reference time; if needed, we could synchronize the phone’s clock to the PC before experiment and then use absolute UTC timestamps for all devices for easier alignment. For now, recording the phone’s uptime timestamp per frame is adequate to get relative timing.

**Ensuring No Data Loss:** The threads approach should ensure we don’t lose frames under normal conditions. If the system is overwhelmed (e.g., very long recordings causing memory pressure), the queues could grow. We will monitor memory usage – e.g., we might set an upper bound on the queue size for streaming; if it’s exceeding, drop frames. For file writing, ideally the storage can keep up with ~2.5MB/s (which it should on UFS 3.0 internal storage easily). A Galaxy S21/S22 can typically write at hundreds of MB/s sequentially, so disk I/O is not the bottleneck; rather, the concern is not to accumulate too much in RAM if the disk hiccups. A simple approach: use a ring buffer of a few frames for file writing – if it ever overruns (which is unlikely), we log a warning and drop the oldest. In testing, we’ll verify that even during 4K recording, the thermal data saving doesn’t lag (the phone’s I/O and CPU are powerful enough for this multi-tasking).

## 8. File Management and Session Integration

The ThermalRecorder must integrate with the session-based organization of recordings. Likely, the app uses a **SessionInfo** object to track a recording session’s metadata (e.g., session ID, start time, participant info, and file paths for each modality). We will use this for naming and storing the thermal data:

* When a session starts, SessionInfo might provide a directory path (for example, /storage/emulated/0/MyAppSessions/Session\_001/). We will create the thermal data file in that directory, e.g. thermal.raw (or thermal.dat). To avoid name collisions, possibly include the session ID or timestamp in the filename. If SessionInfo has a convention (like <sessionName>\_thermal.dat), follow that.
* Ensure that the file is opened at start and **closed on stop**. If the user stops the recording via the UI or if the camera disconnects unexpectedly, ThermalRecorder should gracefully close the file (flush buffers) so that data is not corrupted. Using try-with-resources or finally blocks around the writing loop is a good practice. On stop, also terminate the background threads or let them finish processing any last frames, then release the camera.
* The SessionInfo can also be used to log events: for instance, it might have a function to add a note like “Thermal recording started at [timestamp]” which could be written to a session log. That can help later in aligning data. If such logging is part of the system, we will add entries for thermal start/stop. Additionally, we can include calibration frames if any (for example, if there is a routine to capture a flat-field or shutter event, log it).
* **Directory structure**: Each session folder would contain something like: video.mp4 (RGB video), shimmer.csv (sensor readings), and now thermal.dat (raw thermal data). We should also consider saving a **thermal preview video** if needed. The requirements don’t explicitly ask for a thermal video file (they want raw data instead), but in case someone later wants an easy-to-view thermal video, we might optionally encode the thermal frames into an MP4 (using MediaCodec) in parallel. This wasn’t requested, but our design is open to that (we could reuse the preview frames for encoding a simple MP4 if needed). The primary deliverable, however, is the raw data file.
* **Post-recording handling**: Once recording stops, the phone could notify the PC that a thermal file is ready. Depending on the workflow, the PC might then pull the file (e.g., via the socket or a separate transfer mechanism). Since these files can be large, an immediate transfer might be optional. The plan suggests either manual retrieval or triggered transfer after stop[[35]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=%28because%20streaming%204K%20in%20real,after%20capture%20would%20be%20useful). We won’t implement the transfer as part of ThermalRecorder itself (that likely belongs to a higher-level controller), but ThermalRecorder can report the path/filename to whoever needs it (so that the PC or the app’s UI can act on it).

In summary, SessionInfo integration means ThermalRecorder does not operate in isolation – it uses the session context for file placement and timing. It also should update session state if needed (for example, marking that thermal data is being recorded). If there is a unified control that starts/stops all modalities, ThermalRecorder should register itself with that controller. The design remains modular: the ThermalRecorder can be tested independently by simulating a session info and ensuring it writes to the correct location, then integrated into the full system.

## 9. Performance and Memory Considerations (Samsung S21/S22)

The target devices (Galaxy S21/S22) are high-end phones with powerful processors and UFS 3.x storage, which is encouraging for our multi-modal recording scenario. Still, we must optimize performance to avoid dropped frames or overheating:

* **CPU Load**: Recording 4K video (RGB) uses the hardware encoder (GPU/DSP), so CPU is mostly free for other tasks[[36]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=might%20be%20to%20record%20the,by%20timestamping). The thermal camera SDK processing (image conversion, etc.) does have a CPU cost, but the data size is small. The Infisense SDK is partly native and likely uses NEON optimizations for things like color conversion and temperature calculations. We will keep the CPU load low by using those native methods (as opposed to, say, manually iterating over pixels in Java). Empirically, handling a 256×192 frame 25 times a second (with perhaps a bit of matrix math for temperature) is trivial for the S21’s Snapdragon 888 or S22’s Snapdragon 8 Gen1. Even adding JPEG compression at 10 fps is not significant for these chips. Nonetheless, we avoid unnecessary work – e.g., no creating excessive objects each frame (use object pools or reuse byte arrays, Bitmaps, etc.).
* **Memory Usage**: Thermal frames buffers are on the order of 100-200KB each. Even buffering a few dozen frames won’t dent memory on devices with 8GB RAM. The main memory concerns are avoiding **memory leaks** and not accumulating data indefinitely. We will ensure to release the Bitmap objects in the preview thread if the activity/fragment is destroyed, unregister USB monitors when not needed, and close files. The use of large buffers should be static or pooled. For example, allocate imageSrc and temperatureSrc once at init based on the resolution[[1]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L60-L63), rather than allocating for each frame. The same for the ARGB buffer for display – allocate once (256×192×4 bytes ~ 200KB) and reuse. The only significant memory usage could be the raw file itself growing on disk (which doesn’t affect RAM, but affects storage). We should check available storage before recording and perhaps limit recording duration if space is low. We could integrate a check: if free space is below a threshold, warn the user or stop recording to prevent crashes due to no storage.
* **I/O Throughput**: As calculated, ~2.5 MB/s of thermal data plus the RGB 4K video (which could be ~30-60 Mbps, i.e., 4-7.5 MB/s) are being written simultaneously, plus Shimmer data (negligible kb/s). So total write could be ~10 MB/s worst-case. The internal storage can handle this easily (for reference, modern smartphones often sustain >50-100 MB/s sequential writes). Thus, disk throughput is safe. We should, however, use buffered output streams for writing the thermal file to minimize syscall overhead (Java’s BufferedOutputStream or FileChannel in burst writes). Flushing IO less frequently (maybe every few MB or at stop) can reduce overhead. The downside is if a crash happens mid-run, buffered data might not be written – but since we stop properly, flush at stop. We might call fos.getFD().sync() after closing to ensure data is on disk.
* **Device Thermals**: Running both cameras and Wi-Fi can heat up the phone. The S21/S22 have good cooling but are known to thermal-throttle under sustained heavy CPU+GPU load. Our workload is moderate (video encode is offloaded, thermal processing is light). Still, during long sessions, the phone temperature should be monitored. If the device’s thermal management starts to throttle the CPU, our processing might slow or frames might drop. We can implement a simple temperature monitor (Android provides battery/cpu temp APIs) and log if temperature goes critical. The plan suggests allowing breaks between recordings if needed[[37]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Thermal%2FPerformance%3A%20Recording%204K%20on%20a,processes%20or%20using%20performance%20modes). For now, our code doesn’t automatically stop on overheating, but we will document this as a consideration. The user can be advised to keep the phones cool (perhaps using a fan or AC in the room for very long experiments).
* **Concurrent Camera Access**: One potential issue on some Android devices is opening two cameras at once. Many phones cannot use the internal RGB camera and an external USB camera simultaneously due to bandwidth or hardware constraints. However, on the S21/S22, using the USB camera does not rely on the phone’s Camera2 API (the USB goes through external interface), so it likely bypasses the limitation that Camera2 might have with dual cameras[[38]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=References%3A). We have precedent that the device can show up as a UVC webcam. The Topdon camera essentially becomes a UVC video source over USB, which doesn’t contend with the phone’s camera hardware pipelines. We expect it to work, and indeed Topdon’s app allows using the device while the phone’s normal camera is free. In case of any conflict (for example, if the USB bus bandwidth is an issue with 4K video being saved to USB storage or something), we will test and adjust. The S21 has USB 3.2 gen1 on the go, which should handle the tiny data of the thermal cam easily.
* **Testing on both S21 and S22**: Minor differences might exist (chipset, OS version). We ensure the app requests **USB permission every time** after device connect (Android may remember permission for a device until reboot, depending on OS). If any issues (e.g., on one device the permission dialog doesn’t appear due to timing), we handle it by possibly using the manifest intent filter with device-filter xml to auto-grant if the app is default for that device.

In summary, the design decisions (using background threads, efficient binary formats, hardware encoding, low preview framerate) are all aimed at keeping the system running in real-time without hiccups. We will validate that during the test phase.

## 10. Manual Test Plan

To ensure the ThermalRecorder meets all requirements and performs well, we will conduct a series of manual tests. Below is a structured test plan:

* **Connection and Permission Test:**
* With the app installed on the phone (S21/S22), plug in the Topdon TC001 thermal camera via USB-C.
* Expectation: Android prompts to grant the app permission to use the USB device. Grant the permission.
* The app’s ThermalRecorder (or a portion of the UI) should indicate the camera is connected – e.g., start showing a preview or at least no error.
* Unplug and replug the camera to test the attach/detach handling. The preview should stop on detach and resume (after permission) on reattach.
* **Live Preview Display Test:**
* Once the camera is connected and permission granted, verify that the **live thermal image** appears on the phone’s screen. Point the camera at various scenes (hand, cup of hot water, ice pack) to see that hot and cold areas are visible in the preview. The preview should update smoothly (near-real-time).
* Rotate the phone and camera if possible. If the physical orientation changes (e.g., phone in landscape vs portrait), confirm that the image rotation logic works (the image isn’t sideways or upside-down). We may need to force a particular orientation for the activity running the preview to avoid confusion, or handle rotation via code as implemented.
* Verify that the preview quality is acceptable – no significant lag or tearing. Also check that the image scaling is correct (the aspect ratio of the thermal image should be maintained; circles shouldn’t look oval, etc.).
* **Start/Stop Recording Synchronization Test:**
* Initiate a recording from the PC controller interface (this should send a START command to all devices). Verify that the phone begins recording RGB (the 4K video), starts the Shimmer data stream, **and starts the ThermalRecorder** simultaneously. There might be an indicator or log message confirming each started.
* Let it record for a short duration (e.g., 10 seconds) and then send a STOP command. Verify all three modalities stopped.
* Check that after stopping, the ThermalRecorder closed its file and released the camera properly (the preview might either remain on or could be turned off depending on design – ideally, we keep preview on for convenience). Ensure that a second recording can be started without restarting the app, to confirm proper re-initialization (i.e., do two back-to-back recordings to see no resource conflicts or crashes).
* **Data File Integrity Test:**
* After a recording, use Android file explorer or connect the phone to a PC to retrieve the thermal data file (.dat or .raw). Confirm the file exists in the expected session folder and has a non-zero size.
* Write a small Python or MATLAB script to parse the file: using the known width=256, height=192, read the binary data and reconstruct a frame sequence. Check that the number of frames roughly matches the recording duration × frame rate. For example, if you recorded ~10 seconds, you expect on the order of 250 frames. It doesn’t have to be exact if we throttled or dropped some, but it should be in the ballpark and consistent (if exactly 10s and we didn’t drop frames, it’d be ~250).
* Extract a single frame’s data (e.g., the first frame) and visualize it as an image (plot the matrix as an image with a color map). The thermal structure should be recognizable (e.g., if you pointed at a hand, the hand shape should appear in the temperature matrix). Verify that the temperature values make sense: for instance, find the max and min of the temperature matrix – are they reasonable (e.g., maybe the max was ~30700 in raw units which could correspond to ~30°C if scaled by some factor, etc.). If possible, convert the raw values to actual Celsius using the known formula or by a reference comparison (e.g., compare a spot on the image with a contact thermometer). This validates the radiometric correctness.
* If any frame appears corrupted (e.g., all zeros or random noise), that indicates a problem in our saving logic (perhaps a synchronization issue). In testing so far with short runs, we expect none, but we should test a longer run (see next).
* **Long Duration Performance Test:**
* Conduct a longer recording (e.g., 5 minutes) to test stability. During this, monitor the phone’s temperature if possible (the app could display it or use an Android system monitor). Also monitor if any thermal frames are dropped or if preview slows down over time (which could indicate thermal throttling).
* After 5 minutes, stop the recording. Check that the thermal file is ~5min × 2.5MB/s ≈ 750 MB (if it’s much smaller, maybe frames were dropped or we weren’t recording at full rate). A significantly smaller file might mean we inadvertently were throttling the saving; a larger (impossibly large) file might mean something was mis-calculated, but that’s unlikely.
* Verify the integrity of this file as well (perhaps not frame-by-frame, but check beginning, middle, end frames for sanity). Also ensure the phone remained responsive and didn’t overheat (if the device did get very hot, note if any frame drops occurred near the end or if the system issued any thermal warnings).
* **Concurrent Operation with RGB and Shimmer:**
* During a test recording, pay attention to the RGB video and Shimmer data capture. The goal is that the ThermalRecorder’s activity does not interfere. For example, ensure the 4K video file isn’t getting skips or timestamp issues. After recording, play back the 4K video to see if it’s smooth. Also, check the Shimmer data log to see if there are continuous timestamps (no large gaps that could indicate the phone was too busy).
* If possible, do a **synchronization check**: e.g., have a LED in view of both the RGB and thermal cameras and turn it on/off at a known time, then later verify that the event is simultaneously reflected in the RGB video and thermal data (within expected time sync error). This would confirm that starting both recordings at the same time was effective and the data can be aligned (since we timestamped frames, we can compare those to video frame times).
* **Topdon TC001 Plus Test:**  
  If we have access to the TC001 Plus model, repeat the above **connection and preview tests** with it. The Plus should also be recognized (perhaps as a different PID like 0x5840). Ensure permission and streaming work the same. The plus’s visible-light camera is not explicitly used, but verify that having the plus doesn’t cause any errors in our SDK usage. The radiometric data from plus should be equivalent in format (still 256x192 IR resolution). The presence of the visible lens might internally allow the device to do image-fusion in its own app, but our app will likely just see the IR data. Check that our preview looks like a normal thermal image (not, say, an error or half the image missing because we didn’t handle two channels – the SDK might just ignore the visible channel). Essentially, confirm that IR frames come through on the Plus as well.
* **Edge Cases:**
* **No Camera Connected**: Try starting a thermal recording without the camera attached (or if permission denied). The ThermalRecorder should handle this gracefully – either queue the start until camera is available or immediately error out with a message. It should not crash. This scenario tests error handling.
* **Camera Unplug During Recording**: Disconnect the thermal camera mid-recording. The app should ideally detect the USB disconnect (the USBMonitor onDettach callback) and stop the thermal recording thread safely[[39]](https://github.com/CoderCaiSL/IRCamera/blob/806fbb62ffbfab3418b82d4204bbc0efbbcc68d4/libir-demo/src/main/java/com/infisense/usbir/camera/IRUVCTC.java#L144-L152). The file should be closed properly and the session should mark that thermal ended prematurely. The RGB and Shimmer might continue (since the experiment might choose to keep going). We should test that unplugging doesn’t hang the app. Possibly, we’ll simulate this carefully to avoid corrupting the file. Check the resulting file – it should contain frames up to the point of disconnect and be readable.
* **Multiple Sessions**: Run two sessions in a row and ensure the second session creates a new thermal file (not appending to the old one). SessionInfo should handle unique naming or new directories, but we confirm that data isn’t overwritten or mixed.

By following this test plan, we can validate that the ThermalRecorder meets the requirements: capturing full-frame radiometric data in sync with other modalities, providing live previews locally and remotely, and doing so reliably on the target hardware. The tests also help refine any performance issues (for example, if we notice high CPU, we might adjust the preview frame rate or other parameters as needed).

Overall, the ThermalRecorder module is designed to be **extensible** (e.g., easy to adapt if a new thermal camera or mode is introduced), **modular** (it interfaces cleanly via SessionInfo and a recorder interface so it can be maintained separately), and **efficient** (using native libs and proper threading to achieve concurrency without bottlenecks). By implementing and testing as described, we will achieve the milestone of synchronized thermal video recording alongside 4K RGB and sensor data, fulfilling Milestone 2.3’s objectives.

**Sources:**

1. Multi-sensor recording system plan – outlines parallel RGB and IR capture and preview streaming considerations[[25]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Preview%20Monitoring%3A%20Receive%20live%20previews,to%20switch%20between%20IR%2FRGB%20preview)[[40]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Data%20Streams%3A%20For%20the%20preview,frame%20rate%20preview%20just%20for).
2. Topdon (Infisense) SDK sample code – shows how to integrate the USB thermal camera (permission, dual image+temp frame handling, and conversion for display)[[11]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L291-L300)[[19]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L120-L128).
3. Research discussions on radiometric data – confirm the approach of exporting per-pixel temperature data (e.g., via CSV or binary) for analysis[[31]](https://www.researchgate.net/post/How-can-I-get-the-temperature-at-each-pixel-of-the-IR-image-taken-by-FLIR-T250-camera#:~:text=Hi%2C)[[32]](https://optris.com/us/software/pixconnect/#:~:text=PIX%20Connect%3A%20License,Additionally%2C%20images%20and).
4. System performance notes – recording multiple streams requires efficient use of threads and hardware encoding to avoid overheating or frame drops[[37]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Thermal%2FPerformance%3A%20Recording%204K%20on%20a,processes%20or%20using%20performance%20modes)[[28]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Data%20Upload%2FStreaming%3A%20While%20recording%2C%20the,adb%20pull%20via%20a%20script).

[[1]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L60-L63) [[4]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L110-L119) [[5]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L294-L303) [[8]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L440-L449) [[9]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L444-L453) [[10]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L291-L299) [[11]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L291-L300) [[12]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L301-L309) [[14]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L432-L440) [[20]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L295-L303) [[24]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L20-L23) [[33]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java#L130-L138) IRUVC.java

<https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/src/main/java/com/infisense/usbir/camera/IRUVC.java>

[[2]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L19-L27) [[3]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L196-L203) [[7]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L66-L74) [[17]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L154-L163) [[18]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L164-L171) [[19]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L120-L128) [[21]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L170-L178) [[22]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L174-L181) [[23]](https://github.com/buccancs/topdon-sdk/blob/83329a9fe4ebc275408c872b03aac1f4e13af0b0/ANDROID_SDK_USB_IR_1.3.7/libir_sample/usbir/README.md#L144-L151) README.md

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

[[6]](https://github.com/CoderCaiSL/IRCamera/blob/806fbb62ffbfab3418b82d4204bbc0efbbcc68d4/libir-demo/src/main/java/com/infisense/usbir/camera/IRUVCTC.java#L61-L69) [[39]](https://github.com/CoderCaiSL/IRCamera/blob/806fbb62ffbfab3418b82d4204bbc0efbbcc68d4/libir-demo/src/main/java/com/infisense/usbir/camera/IRUVCTC.java#L144-L152) IRUVCTC.java

<https://github.com/CoderCaiSL/IRCamera/blob/806fbb62ffbfab3418b82d4204bbc0efbbcc68d4/libir-demo/src/main/java/com/infisense/usbir/camera/IRUVCTC.java>

[[13]](https://www.topdon.us/products/tc001-plus?srsltid=AfmBOoo8fJ1BiHiNXsNYkOI6c468tmrv4hOuPYKnV6UDjQrLnYg1rCOQ#:~:text=0) [[15]](https://www.topdon.us/products/tc001-plus?srsltid=AfmBOoo8fJ1BiHiNXsNYkOI6c468tmrv4hOuPYKnV6UDjQrLnYg1rCOQ#:~:text=256x192) [[16]](https://www.topdon.us/products/tc001-plus?srsltid=AfmBOoo8fJ1BiHiNXsNYkOI6c468tmrv4hOuPYKnV6UDjQrLnYg1rCOQ#:~:text=Sporting%20an%20ultra,display%2C%20making%20it%20easier%20to) TC001 Plus (Android Devices) – TOPDON USA

<https://www.topdon.us/products/tc001-plus?srsltid=AfmBOoo8fJ1BiHiNXsNYkOI6c468tmrv4hOuPYKnV6UDjQrLnYg1rCOQ>

[[25]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Preview%20Monitoring%3A%20Receive%20live%20previews,to%20switch%20between%20IR%2FRGB%20preview) [[26]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=sending%20full%204K%20raw%20frames,frame%20rate%20preview%20just%20for) [[27]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Network%20Sockets%20or%20HTTP%3A%20For,frames%20as%20they%20become%20available) [[28]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Data%20Upload%2FStreaming%3A%20While%20recording%2C%20the,adb%20pull%20via%20a%20script) [[29]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=sending%20full%204K%20raw%20frames,the%20operator%20sees%20a%20near) [[30]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=preview%20frames%20,adb%20pull%20via%20a%20script) [[34]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=synchronize%20the%20two,aligns%20with%20the%20PC%E2%80%99s%20timeline) [[35]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=%28because%20streaming%204K%20in%20real,after%20capture%20would%20be%20useful) [[36]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=might%20be%20to%20record%20the,by%20timestamping) [[37]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Thermal%2FPerformance%3A%20Recording%204K%20on%20a,processes%20or%20using%20performance%20modes) [[38]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=References%3A) [[40]](file://file-9JgS9hNU2GwaXbC4UsQQGa#:~:text=Data%20Streams%3A%20For%20the%20preview,frame%20rate%20preview%20just%20for) Updated\_Plan\_for\_Multi\_Sensor\_Recording\_System\_Android\_+\_PC.docx

<file://file-9JgS9hNU2GwaXbC4UsQQGa>

[[31]](https://www.researchgate.net/post/How-can-I-get-the-temperature-at-each-pixel-of-the-IR-image-taken-by-FLIR-T250-camera#:~:text=Hi%2C) How can I get the temperature at each pixel of the IR image taken by FLIR T250 camera? | ResearchGate

<https://www.researchgate.net/post/How-can-I-get-the-temperature-at-each-pixel-of-the-IR-image-taken-by-FLIR-T250-camera>

[[32]](https://optris.com/us/software/pixconnect/#:~:text=PIX%20Connect%3A%20License,Additionally%2C%20images%20and) PIX Connect: License-free Software for IR Cameras - Optris

<https://optris.com/us/software/pixconnect/>