# Milestone 2.8: Calibration Capture and Sync Features – Technical Implementation Guide

## Goals and Overview

The goal of **Milestone 2.8** is to enhance the Android application with special capture modes and synchronization aids for calibration and multi-device time alignment. Specifically, we will implement:

* **Calibration Frame Capture** – On receiving a calibration trigger (from the PC or via a test UI button), the app will capture a synchronized image from the phone’s RGB camera and the attached thermal camera. These paired images (visible and thermal) will be saved locally with matching identifiers (e.g., calib\_001\_rgb.jpg and calib\_001\_thermal.png) for later calibration processing.
* **LED Flash / Sync Signal (Optional)** – Provide a mechanism (if needed) to produce a synchronization signal (like a quick flash of the phone’s flashlight or an audible beep) to aid in aligning recordings across devices. This can be used in setups where a common visual or audio cue helps synchronize multiple video streams.
* **Timestamping and Clock Sync** – Implement a simple clock synchronization with the PC (master device) so that all recorded data (videos from multiple phones, PC webcam, sensor logs) can be aligned to a common timeline. The app will adjust or record timestamps based on a PC-provided reference time to ensure all devices share a consistent time base.

By the end of this milestone, the Android app should be capable of capturing dual-camera calibration images on-demand, optionally provide a sync flash/beep, and tag data with synchronized timestamps. This prepares the system for multi-camera calibration and coordinated recordings across devices.

## System Design Overview

To achieve these goals, we will extend the existing Android app architecture with additional components and methods:

* **Network Command Handling:** The app already communicates with a PC controller (likely via a socket or similar IPC). We will extend the command protocol to include new message types such as CALIBRATE (to trigger calibration capture), SYNC\_TIME (to adjust clocks), and possibly FLASH or BEEP for sync signals. A central command handler will parse incoming commands and dispatch actions.
* **Visible-Light Camera Module:** This controls the phone’s built-in RGB camera (via Camera2 API). We will add functionality to capture a high-resolution still image on demand *even during preview or video recording*. This may involve using a dedicated ImageReader for JPEG images and issuing a TEMPLATE\_STILL\_CAPTURE request. The camera module will handle focus/exposure if needed, capture the image, and provide the image data (or save it to file).
* **Thermal Camera Module:** This interfaces with the Topdon IR camera (via the Topdon SDK). Typically, the thermal camera continuously streams frames. We will modify/extend this module to allow grabbing a frame for calibration when triggered. The implementation might use the next available frame from the stream as the “snapshot” or call a specific SDK method if available. The thermal frame will then be saved as an image (likely PNG format for lossless quality).
* **Calibration Capture Manager:** We may introduce a helper component (or just a coordinated function) that orchestrates simultaneous capture from both cameras. It will ensure both the RGB and thermal captures occur in quick succession, tag them with a common ID or timestamp, and store the image pair. The manager will also handle any preview pause/resume logic, and queueing of captures to avoid conflicts.
* **Sync/Clock Manager:** A lightweight utility to maintain a clock offset between the phone and the PC. When a SYNC\_TIME message is received (containing the PC’s current timestamp), the app will compute the offset (difference between PC time and phone’s current System time). This offset will be used when stamping files or log entries so that all devices use the PC’s timeline (e.g., timestamps relative to PC’s start time or absolute epoch). This manager will also handle periodic re-synchronization if the PC sends multiple sync messages.
* **UI and Configurations:** For testing and flexibility, we’ll include a developer UI option (e.g., a button) to manually trigger a calibration capture on the phone. We will also ensure any required permissions (camera, storage, flashlight, audio) are accounted for in the Android manifest or requested at runtime. IDE (Android Studio) project settings should be updated to include any new SDK references or libraries (if the Topdon SDK wasn’t already included, ensure it’s integrated and recognized by the project).

Overall, the design emphasizes **minimizing delay** between the RGB and thermal capture, and maintaining a **consistent naming and timestamping scheme** for all captured data.

## Implementation Steps

Following is a step-by-step guide to implement the features for Milestone 2.8:

### 1. Extend Command Handling for Calibration and Sync

**Goal:** Enable the app to respond to new control commands (CALIBRATE, SYNC\_TIME, etc.) from the PC (or from the UI for testing).

* **Locate the Network Listener:** Identify where in the app the incoming messages from the PC are handled. This might be in a networking service or thread (e.g., a SocketListener or CommandHandler class). It likely reads strings or packets and uses a conditional or switch-case to determine the command.
* **Add Calibration Command:** Introduce a new case for a calibration trigger. For example, if the protocol is text-based, the PC might send a string like "CALIBRATE" or "CAPTURE\_CALIB". Add logic to handle this command by calling a new function, e.g., onCalibrationCommandReceived().
* *UI Trigger (for testing):* Also add a manual trigger in the app UI. For instance, add a button in the main activity (perhaps labeled "Capture Calib Frame") which when clicked calls the same onCalibrationCommandReceived() method. This is useful to test the functionality without needing the PC each time.
* **Add Sync Time Command:** Likewise, handle a SYNC\_TIME message. Decide on the message format, for example: "SYNC\_TIME:<pc\_epoch\_ms>" or a JSON with a timestamp. Parse the incoming time value (which presumably is the PC’s current Unix timestamp in milliseconds, or a session-relative timestamp).
* When received, call a method like onSyncTimeReceived(pcTime) to process it (we will implement this in Step 5).
* **Optional Flash/Beep Command:** If a sync signal is desired via command, define something like "FLASH" or "BEEP" message. The handler should call the corresponding function to flash the LED or play a beep sound (discussed in Step 4).
* **Threading Consideration:** Ensure that these command-handling callbacks are executed on a suitable thread. If the networking is on a background thread, it might be fine to trigger camera actions directly, but often camera operations must run on the main (UI) thread or a dedicated camera thread. You may need to use an Android Handler or runOnUiThread to forward the action to the main thread (especially for UI or Camera2 API calls).
* **Logging:** Add log statements for each new command for easier debugging. For example: “Received CALIBRATE command from PC” etc., to verify that the commands are being received and parsed correctly.

### 2. Implement the Calibration Capture Routine

**Goal:** Capture a still image from the phone’s RGB camera and a frame from the thermal camera nearly simultaneously, then save them with a common identifier.

We will implement a method performCalibrationCapture() (either in a dedicated CalibrationManager or within the main activity/controller) that does the following:

**2.1 Prepare for Capture:** - *Pause/Freeze Preview (optional):* If needed, pause the live preview UI to avoid any visible glitches. This could mean stopping any continuous updates of the preview Surface for a moment. This step is optional – Camera2 can capture a photo while preview is ongoing – but pausing can ensure the preview doesn’t update mid-capture. If pausing, ensure to resume later. - *Lock Camera Settings (optional):* For better image quality, you may want to lock focus and exposure on the main RGB camera before capturing. For instance, if using Camera2 API: - Use the preview CaptureRequest to lock CONTROL\_AF\_TRIGGER (autofocus) to AF\_TRIGGER\_IDLE or use a continuous focus mode so that the image is not blurred. - Lock AE (auto-exposure) if needed to avoid flicker. (Since calibration targets are likely static, this might not be critical.) - These steps can be done a few frames before capturing to stabilize the image.

**2.2 Capture RGB Photo (Visible Light Camera):** - Use the Camera2 API to take a still image: - Ensure you have an ImageReader configured for high resolution JPEG output. If the app currently only shows preview/video, you might need to add an ImageReader (for example, at the maximum supported resolution for stills) and include it in the camera capture session outputs. - If the camera was opened with a TEMPLATE\_RECORD (for video) or TEMPLATE\_PREVIEW, you can still perform a still capture. You might need to create a new CaptureRequest with TEMPLATE\_STILL\_CAPTURE and target the ImageReader’s surface. - If an ImageReader for stills was not initially configured, you may need to configure one on the fly. (Preferably, configure it when you set up the camera to avoid delays later.) - Build a capture request:

CaptureRequest.Builder captureBuilder = cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE\_STILL\_CAPTURE);  
captureBuilder.addTarget(stillImageReader.getSurface());  
// Optionally, add any settings like auto-focus, orientation, etc.

Use the camera’s current state for focus/exposure (or the locked settings from above). - Submit the capture request to the camera’s CameraCaptureSession with session.capture(captureBuilder.build(), captureCallback, backgroundHandler). - In the ImageReader’s OnImageAvailableListener, retrieve the Image (in JPEG format). Save this image to file: - Generate a unique calibration ID or timestamp. For example, use an incremental counter or a timestamp (like calib\_1630429093000). This will be used in filenames. - Construct file path for RGB image, e.g., calib\_<ID>\_rgb.jpg. - Write the Image’s byte buffer to the file (ensure to close the Image after saving to free memory). - *Note:* JPEG output will already be encoded. If using another format, you might need to encode or convert to PNG. But JPEG is fine for the RGB photo. - *Handling Camera States:* After capturing, if the preview was halted or if the camera’s AE/AWB/AF were locked, remember to unlock or resume: - Resume the continuous preview by sending the normal repeating request to the camera session again (if it was stopped). - Unlock focus/auto-exposure if they were locked (send triggers to cancel the locks if applicable). - Be mindful of the slight pause that might occur during capture. Users might notice a quick freeze when the picture is taken – this is normal.

**2.3 Capture Thermal Image (Infrared Camera):** - The thermal camera likely provides frames continuously via a callback (e.g., a listener that provides a bitmap or byte array for each frame). We will utilize the next available frame as the calibration image: - Set a flag like pendingCalibCapture = true in the thermal camera module when calibration is triggered. In the thermal frame callback (where each new thermal frame arrives), add logic such as:

if (pendingCalibCapture) {  
 pendingCalibCapture = false;  
 Bitmap thermalBitmap = ... // get the current frame as bitmap or data  
 saveThermalImage(thermalBitmap, calibID);  
}

This ensures that as soon as the next frame comes in, we grab it and save it. The delay between triggering and frame capture should be very small (on the order of the frame interval, e.g., 30-60ms if thermal is ~15-30 FPS). - If the SDK allows an explicit snapshot call (for example, some IR cameras have a method to capture a still image), you can use that. But often, using the next frame is simplest and ensures we don’t interrupt the streaming. - **Saving the Thermal Frame:** When the frame is captured: - Use the same calibration ID generated in step 2.2 for the filename (e.g., calib\_<ID>\_thermal.png). - If the thermal frame is already a Bitmap or byte array in a standard format (RGB/BGR values), you can encode it to PNG or JPEG. PNG is preferred for thermal since it’s lossless (better for later analysis of pixel values). - Use Android Bitmap compression or an image I/O library to write the PNG file. For example:

FileOutputStream fos = new FileOutputStream(thermalFile);  
thermalBitmap.compress(Bitmap.CompressFormat.PNG, 100, fos);  
fos.close();

Ensure proper error handling and that the storage location is writable (more on storage setup below). - **Coordinate Timing:** The RGB capture and thermal capture are initiated almost together. We expect the thermal “next frame” to align closely with the moment the RGB photo was taken. By tagging both images with the same ID and system timestamp, the calibration algorithm on PC can treat them as a pair. - **Confirmation/Callback:** After both images are saved, you might: - Send a confirmation back to the PC (e.g., a message like CALIBRATION\_DONE:<ID>). - Or simply log locally that calibration frame <ID> is captured. The PC could later request the images or pull them from the device storage as needed. - **Edge Cases:** Consider if the user triggers calibration in rapid succession: - Use a thread-safe way to generate unique IDs (atomic counter or timestamp) and to handle the possibly overlapping operations. Ideally, disable a new calibration trigger until the current one finishes capturing both images. - If the cameras are busy (e.g., the main camera might still be processing a capture), you might queue the request or show a brief busy indicator until ready.

### 3. Save Images and Manage Storage

**Goal:** Define where and how the calibration images are stored, and ensure the app has permissions to do so.

* **Storage Location:** Decide on a directory to save calibration images:
* A convenient choice is the app’s external storage directory (which is accessible when the device is connected to a PC, making it easy to retrieve files). For example: Documents/IRCameraCalibrations/ or Pictures/IRCamera/Calibration/.
* Alternatively, use internal app storage and implement a method to transfer images to PC via the network. However, for initial implementation, external storage (or MediaStore) is simpler for access.
* **Filename Convention:** As mentioned, use a common ID. For example:
* RGB image: calib\_<session>\_<index>\_rgb.jpg (session could be a date or session ID if multiple sessions; index is the calibration shot number in that session)
* Thermal image: calib\_<session>\_<index>\_thermal.png
* Including a timestamp in the metadata (EXIF for JPEG or a separate log entry) can also be useful. We might embed the synchronized timestamp (from Step 5) in the filename or keep a sidecar file mapping IDs to PC timestamps.
* **Permissions:** Writing to external storage on modern Android (API 30+) may require either using the MediaStore API or requesting WRITE\_EXTERNAL\_STORAGE permission (if targeting API 29 or lower, or using legacy storage).
* Update AndroidManifest.xml with <uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" /> and corresponding read permission if needed.
* If required by the API level, request the permission at runtime before the first time saving images.
* Alternatively, use getExternalFilesDir() which gives an app-specific external path that doesn’t require user permission and is still accessible via USB (within Android/data/...). This might be a good compromise.
* **Ensure Sufficient Space:** Calibration images (especially RGB photos) can be several MB each. This is usually fine, but be mindful if taking many calibrations. Clean up old images or limit the number stored if necessary.
* **Verify Save Functionality:** After implementing, run a quick test (e.g., trigger a calibration, then use Android Studio’s Device File Explorer or a file manager on the phone to check the image files). Verify that both images are written correctly and have reasonable file sizes and formats.

### 4. Implement Optional Sync Signal (Flash/Beep)

**Goal:** If precise visual or audio synchronization cues are required, implement a feature to flash the device flashlight or play a sound on command.

This step may be optional if time synchronization (Step 5) suffices, but we document it for completeness:

* **Flash the LED:** Using the phone’s camera flash can provide a visual cue:
* Acquire a handle to the camera flash. If using Camera2 API for the RGB camera, you can use a CaptureRequest to trigger the flash or use the CameraManager to toggle torch mode.
  + Simple method: Use CameraManager.setTorchMode(cameraId, true) to turn on the flashlight (requires API 23+). After a short delay (e.g., 100ms), turn it off. This produces a blink. Make sure to get the correct cameraId for the device’s back camera with flash (you can obtain the list of camera IDs and check the one with flash capability).
  + Note: The app must have android.permission.CAMERA (which it likely already does) to control the torch. No separate flashlight permission is needed for setTorchMode (on most devices).
  + Also ensure the camera is not in use by another process that prevents torch (but since our app is using it, it should allow torch mode if we haven’t locked it exclusively in a conflicting way).
  + Alternatively, if the camera is open, you can send a capture request with FLASH\_MODE\_TORCH or FLASH\_MODE\_SINGLE to simulate a flash. However, that typically flashes during capture. Using the torch directly as above is simpler for a manual flash cue.
* Implement a function flashSyncSignal() that turns on the torch, waits 0.1-0.5 seconds, then turns it off. This could be triggered on a "FLASH" command or even automatically at start of recording if desired.
* **Audio Beep:** In case an audible sync is useful (e.g., devices can record audio and we want a sync spike in audio waveform):
* Use Android’s ToneGenerator or SoundPool to play a short beep (like 1000 Hz tone for 200ms) at maximum volume. This requires no special permission for playback.
* Ensure the media volume is not muted on the device. You might allow the user to configure this or just document that volume should be up for sync beeps.
* **When to Use:** In practice, if all devices are triggered by the PC simultaneously and we have clock sync, a flash/beep may not be strictly necessary. However, if you observe any drift or need a precise alignment in post-processing, having a flash in the video frames (for cameras that can see it) or a clap sound in audio can help manual or software sync (like how a clapperboard works in film production).
* **Testing Flash/Beep:** After implementing, test by sending the command (or tapping a test UI control):
* For flash: Darken the room slightly and visually confirm the phone’s flashlight blinks.
* For beep: Listen for the tone. Optionally, if other devices are recording audio, check their audio tracks for the beep signal spike.
* Ensure these actions do not crash the app or interfere with ongoing camera operation (a brief torch on shouldn’t disrupt the Camera2 session, but if it does, consider using the screen flash method as an alternative).

### 5. Implement Time Synchronization (Clock Offset)

**Goal:** Align the phone’s timestamps with the PC’s timeline by computing an offset, ensuring all logged data can be related in time across devices.

* **Message Format:** As determined in Step 1, the PC will send a sync message. For example: SYNC\_TIME:1630429123456 (where the number is the PC’s current Unix time in milliseconds, or a reference start time). It could also be a custom epoch (like experiment start = 0). Clarify this with the PC-side implementation.
* **Compute Offset:** In onSyncTimeReceived(pcTime):
* Get the phone’s current system time in the same units (likely System.currentTimeMillis() for Unix epoch in ms).
* Compute offset = pcTime - phoneTime. This offset (which may be positive or negative) tells us how to convert phone times to PC times. For example, if offset = 500 ms, it means phone’s clock is 0.5 seconds behind PC’s clock, so we need to add 500 ms to phone timestamps to get the PC time.
* Store this offset in a globally accessible place, e.g., a static variable in a TimeSyncManager or in the Application class. **All devices should ideally share the same reference clock (PC)**, so each phone will have its own offset value.
* **Utilize Offset for Timestamps:**
* When recording videos or saving files, use the offset to label times. For instance, if the app writes a timestamp into a video file’s metadata or names a file with a timestamp, adjust it: adjustedTime = phoneTime + offset.
* For our calibration images, we could include the PC time in the filename or in a separate log:
  + e.g., after capturing, we know phone capture time T\_ph (maybe when the command was received or when the image was taken). Compute T\_pc = T\_ph + offset, and perhaps write a small .txt file or log entry: calib\_<ID>: PC\_time=<T\_pc>.
  + This can help later to match calibration images to other data (like which frame in a video they correspond to).
* If the app logs sensor data or other events, likewise adjust those times.
* **Periodic Sync:** Clocks can drift over long sessions. If needed, the PC might send SYNC\_TIME periodically (e.g., every few minutes). Each time, recalculate and update the offset. You might smooth abrupt changes if necessary (averaging offsets), but if all devices run NTP-synced clocks, drift should be minimal in short term.
* **Network Latency Consideration:** The above method assumes instant reception of the sync message. In reality, network delay (like 10-50 ms over Wi-Fi) could introduce error. For greater accuracy:
* Implement a round-trip sync: Phone receives SYNC\_TIME (with PC timestamp T\_pc\_send), phone responds immediately with its own timestamp, PC compares to its current time to estimate latency and offset. This is akin to an NTP sync algorithm. However, this complexity might be overkill for our use. Our use-case tolerates tens of milliseconds difference given video frame rates.
* If high precision is needed, one could incorporate this, but to keep it simple, we will trust a single direction sync and assume network latency is low on a local network.
* **Testing Clock Sync:** You can test whether the sync works by:
* Logging the offset value on the phone when sync is received (e.g., print “Clock offset set to X ms”).
* Manually compare phone’s clock and PC clock to see if X roughly equals the difference.
* If possible, have PC send a sync, then immediately have the phone echo back a message with its current time and PC time (post-offset) to verify alignment.
* In practice, use the offset in a scenario: e.g., start a recording on PC and phone, then see if events (like a flash or motion) have correct relative timestamps. This is more of an end-to-end test with the whole system.

## Project Setup and Configuration

Before running and testing these features, ensure the following project configurations:

* **Permissions in AndroidManifest:** Double-check that you have the necessary permissions declared:
* <uses-permission android:name="android.permission.CAMERA" /> (for camera use – likely already present).
* <uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" android:maxSdkVersion="28"/> (if targeting Android 10 or below and using direct file writes) and possibly <uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE" android:maxSdkVersion="28"/>. If targeting API 29+, consider using <uses-permission android:name="android.permission.MANAGE\_EXTERNAL\_STORAGE" /> or MediaStore approach for file saving. For simplicity, during development you can target an SDK < 30 or use app-specific directories.
* <uses-permission android:name="android.permission.FLASHLIGHT" /> is typically not needed for flashlight control (it’s a system permission), just ensure camera permission.
* <uses-permission android:name="android.permission.RECORD\_AUDIO" /> if you plan to use microphone for any audio sync (beep detection) – not strictly needed just to play a beep.
* **Topdon SDK Integration:** The project should include the Topdon (Infisense IRUVC) SDK library. Verify that the SDK’s native libraries and Java classes are properly referenced in your Gradle configuration and that the app can already receive thermal frames. No new dependency is needed for the SDK if it’s already working for streaming.
* **Threading and Handlers:** Ensure you have a background handler thread for camera operations if required:
* For Camera2, often a HandlerThread is used for camera background tasks (like processing images). If you have one (e.g., cameraBackgroundThread), use it for the ImageReader callback to save images off the main thread.
* For receiving network messages, if on a background thread, you might need to communicate with the main thread for UI or camera triggers. Use a Handler or Android’s LiveData/EventBus (depending on app architecture) as appropriate.
* **IDE Configuration:** Open Android Studio and sync the Gradle project after adding any new permissions or SDK references. No special IDE setup is needed beyond standard; just ensure your development device is connected:
* The device should have the external thermal camera connected (via USB-OTG) if applicable.
* If using two phones for testing multi-device sync, you might install the app on both.
* **Build & Install:** Rebuild the app to ensure no syntax or integration errors. Install the updated app on the test device.

## Testing and Verification

With implementation done, follow these checkpoints to test and verify each feature:

1. **Calibration Capture Test (Single Device):**
2. Launch the app on the phone (with the thermal camera connected and streaming, and the RGB camera preview active).
3. Use the new **Calibration** trigger:
   * If using the test UI button, tap the "Capture Calib Frame" (or equivalent) button.
   * If using PC command, send the CALIBRATE command from the PC software and ensure the phone receives it (watch logs).
4. Observe the app behavior: the preview might pause briefly and resume. Within a second or two, the calibration should complete.
5. Using a file explorer (or Android Studio’s device file explorer), navigate to the chosen storage directory (e.g., /Pictures/IRCamera/Calibration/ or app external files directory) on the phone.
6. Verify that two new image files are saved: one for RGB (\*.jpg) and one for thermal (\*.png), with the same ID in their names. Check the timestamp or index in the filename to ensure they match.
7. Open the images to verify content: They should depict the same scene/moment from the two cameras. For example, if you had a checkerboard or calibration pattern in view, confirm it’s visible in both (the thermal image may show a heat signature if applicable).
8. Repeat the test for multiple triggers (take several calibration shots). Confirm that each time, new files are created with unique IDs (e.g., incrementing) and no overwriting occurs. Also verify the app remains stable after multiple captures (no crashes or memory leaks).
9. **LED Flash Sync Test (if implemented):**
10. In a slightly dim environment, send the flash command (either via PC or a test button).
11. Visually confirm the phone’s flashlight turns on briefly and then off.
12. If another camera (or the PC’s webcam) is observing the phone, check that the flash was visible in that footage (this confirms the usefulness of the flash for sync).
13. Test edge cases: Try flashing while the camera is recording or right after a calibration capture to ensure it doesn’t conflict. The flash should ideally not disrupt the camera preview or thermal feed (a very brief pause might occur if using the same camera, but it should recover).
14. **Audio Beep Sync Test (if implemented):**
15. Send the beep command or trigger the function. Listen for the sound. Ensure it’s audible.
16. If multiple devices are recording audio (say two phones or PC mic), later verify that the beep is present in all recordings at the same point in time (this might involve comparing audio timelines, which can be done in post-processing).
17. Adjust the volume/tone duration if needed based on audibility or clarity in recordings.
18. **Time Sync and Timestamp Verification:**
19. Start with clocks unsynchronized: maybe manually set the phone’s time a minute off from the PC (if possible, or just note the difference).
20. Have the PC send a SYNC\_TIME command with its current time. Check the app’s logs or UI (if you display it) for the calculated offset. It should roughly equal the difference between the clocks.
21. Immediately after syncing, trigger an event from both PC and phone:
    * For example, PC writes a log “Event A at <pc time>” when sending a command, and phone upon receiving writes “Event A at <pc time computed>”. Compare the two timestamps to ensure they match or are within a small margin.
22. Alternatively, after sync, perform a simultaneous action: e.g., flash the phone’s LED (with phone logging the PC-based timestamp) and record with PC’s webcam noting its time. Check if the flash frame aligns with the logged time.
23. Over a longer session, test sending sync at start and end:
    * Compare the offsets to see if there was drift.
    * If drift is significant (more than, say, 50ms over time), consider increasing sync frequency or investigating device clock stability.
24. **Integration Test (Full Scenario Simulation):**
25. Simulate a full recording session with the PC and multiple phones:
    * Start all devices, connect them to the PC control.
    * Send a global start command (from previous milestones, presumably) to begin recording video on all, including PC.
    * During recording, send a calibration command at least once. Phones should capture calibration frames without stopping the recording (the preview might pause momentarily but video recording should ideally continue if configured properly).
    * Optionally trigger a flash/beep at start for manual sync reference.
    * Stop recording.
26. After the session, collect all outputs: videos from phones, images from calibration, etc. Verify:
    * Calibration images pairs are saved and can be matched to the video frames (e.g., find the timestamp in video where calibration happened – the calibration might not appear in video if it was just a still capture, which is fine; the images are separate outputs for processing later).
    * All data has timestamps that line up (using the sync offsets). For instance, if Phone A video says an event happened at t=10s (PC time) and PC’s own data says t=10s, they truly correspond.
    * No data loss or crashes occurred during this intensive test.
27. **User Experience Check:**
28. Ensure that none of these features adversely affect the user’s experience when not in use. For example, if not using calibration mode, the app should operate as before without interruption.
29. The calibration capture should be quick (almost instant from user perspective) and not require them to manually do anything on the phone aside from initiating it.
30. If applicable, update any on-screen indicators: maybe show a brief message “Calibration image saved” or a flash on the screen to indicate success, so the user knows it happened.

## Class and Module Breakdown

To clarify the implementation, here is a breakdown of modules/classes (existing and new) and their roles in Milestone 2.8:

* **Networking/Command Module** (e.g., CommandServerThread or SocketClient):  
  *Role:* Listens for incoming messages from PC.  
  *Updates:* Will include new cases for CALIBRATE, SYNC\_TIME, FLASH, etc. This module triggers the appropriate actions in other modules (calling calibration capture, sync handling).
* **Main Camera Controller** (e.g., RgbCameraManager using Camera2 API):  
  *Role:* Manages the phone’s RGB camera (opening, preview, video, etc.).  
  *Updates:* Add capability to capture a still photo on demand. Possibly maintain an ImageReader for JPEG. Provide a method captureStillImage(callback) that the Calibration manager can call. Ensure this controller can handle the capture while preview or recording is ongoing (Camera2 supports concurrent recording and still capture if configured properly).
* **Thermal Camera Controller** (e.g., ThermalCameraManager leveraging Topdon SDK):  
  *Role:* Interfaces with the external IR camera, receives frames continuously.  
  *Updates:* Provide a method to fetch or receive a single frame for calibration. For example, a flag or listener adjustment as described to save the next incoming frame. Ensure thread-safety if frames come from a separate thread (the SDK might have its own thread/callback for frames). Possibly include a routine to convert frame data to a Bitmap if the SDK gives raw data.
* **CalibrationCaptureManager** (new, or integrated in an existing coordinator class):  
  *Role:* Orchestrates the dual capture process.  
  *Implementation:* Could be a simple function in the main activity or a standalone helper class. It will call RgbCameraManager.captureStillImage() and simultaneously coordinate with ThermalCameraManager to get the next frame. Responsible for generating the calibration ID and calling file save routines. Also could handle notifying the network module that calibration is done.  
  *Note:* If implemented as a class, it might hold state like the last calibration ID, etc. If just as methods, ensure to pass necessary parameters.
* **SyncManager/ClockSyncUtil** (new utility class or just part of network handling):  
  *Role:* Stores the time offset and provides methods to get synchronized time.  
  *Implementation:* For example, a singleton or a static util with setOffset(ms) and getSyncedTime() that returns System.currentTimeMillis() + offset. This can be used throughout the app when timestamps are needed. Keep the offset in memory (could also persist if needed, but since it can change per session, in-memory is fine).
* **UI Components:**  
  *Role:* Buttons or indicators for testing calibration and showing status.  
  *Updates:* Add a button for manual calibration trigger in the UI (for development use). Possibly an on-screen indicator when a calibration photo is taken (e.g., a quick flash overlay or a toast message “Calibration Captured”). This gives feedback to the user. Ensure the UI has necessary context to call into the calibration routine (likely via the main activity or a ViewModel, depending on architecture).
* **Storage/Media Module:**  
  *Role:* Handles writing image files and possibly managing file naming.  
  *Implementation:* Could be simple static methods or part of calibration manager. For example, a File saveImage(ByteBuffer jpegData, String filename) for RGB and a similar for thermal Bitmap. Use Android’s file APIs as discussed. If images need to be accessible in gallery, consider adding them to MediaStore (not mandatory for our case, since PC will retrieve them directly).

Ensure all these parts are properly connected: e.g., the Network module knows where to call for calibration (it might call a method in the main activity or a central controller). If using an MVP/MVVM architecture, the command could emit a LiveData event that the UI layer observes to trigger the capture. In a simpler approach, if everything is in an Activity/Service, direct method calls are fine.

## Additional Tips and Considerations

* **Synchronization Accuracy:** While we aim for near-simultaneous capture, perfect hardware sync is usually not possible with two separate cameras. The expected tiny delay (perhaps 50ms or less difference) between the RGB and thermal capture is usually acceptable for calibration purposes (especially if the subject is static). If a more precise sync is needed (for example, for moving subjects), consider capturing a *brief video* from both cameras and aligning via post-processing – but that’s beyond our current scope.
* **Thermal Camera Behavior:** Some thermal cameras have a shutter calibration (NUC) that happens periodically (you might notice a click in the device and a pause in frames). If a calibration command comes exactly at that moment, the thermal frame might be briefly unavailable. You might handle this by detecting if no frame arrives within a short time window and retrying once the thermal resumes. In practice, just be aware of this possibility.
* **Resource Management:** Capturing high-res images and saving to disk is memory and I/O intensive:
* After capturing, free up the Image objects and Bitmaps to avoid memory leaks (e.g., close ImageReader images, recycle Bitmaps if used).
* If many calibration captures are done, monitor that performance doesn’t degrade (should be fine if each is a separate capture).
* **IDE Debugging:** Use breakpoints or logging in Android Studio to step through the calibration capture sequence the first time. This can help catch any thread issues (for instance, trying to start a capture from a non-allowed thread) or file permission problems early.
* **User Notifications:** If this app is used by end-users (not just developers), consider adding user-facing messages or UI elements when calibration happens. E.g., “Captured calibration image #3”. This feedback can be important if the user is manually initiating calibration. Since the PC likely controls it, this is less critical, but still helpful for awareness.
* **Future PC Integration:** The PC software will eventually need to retrieve these calibration images. You can plan for how that might happen:
* Perhaps a command where the phone sends the image files over the socket (as bytes). This could be heavy but feasible for a few images.
* Or simply instruct the user to manually copy them. Given we’re saving on the phone, ensure the path is known. Possibly, document that the images are saved under IRCamera/CalibrationShots on the device.
* For now, focus on capture; the transfer mechanism can be decided later (maybe in another milestone or when integrating PC side).
* **Testing on Multiple Devices:** If you have two Android phones available, install the app on both and try simultaneous calibration commands:
* The PC could send one CALIBRATE that both phones react to. Then you’d have two sets of images. Verify each phone saved its own images correctly.
* This is the real use-case: calibrating multiple camera pairs. For example, Phone A’s thermal vs Phone A’s RGB, and perhaps Phone B similarly. Each pair on each device should be internally synced, and all stamped in the same timeline via the clock sync. This allows correlating everything later.

By following this guide step-by-step, you will implement the calibration capture and sync features robustly. This will significantly enhance the multi-device recording system by providing the necessary data for camera calibration (aligning thermal and visible imagery) and ensuring that all devices’ data can be synchronized in time during analysis. Proceed with coding each part carefully, and make use of logs and tests at each checkpoint to validate the functionality. Good luck with the implementation of Milestone 2.8!