# Implementation Guide: CameraRecorder Module (Milestone 2.2)

## 1. CameraRecorder Class Structure and Public API

The CameraRecorder will be a self-contained class managing camera capture sessions. It should provide a clear **public API** for the app to control recording sessions. Key aspects of the class structure include:

* **Public Methods:**
* initialize(TextureView): Prepares the camera (selecting the appropriate camera and outputs) and binds a TextureView for live preview. This may also start background threads or coroutines for camera operations.
* startSession(boolean recordVideo, boolean captureRaw): Starts a capture session based on flags to record 4K video, capture RAW images, or both. It should configure outputs (MediaRecorder, ImageReader) accordingly and begin the camera preview (and recording if enabled). Returns or updates a SessionInfo object with details (file paths, start time, etc.).
* stopSession(): Stops any ongoing recording, flushes pending captures, and releases camera resources. Ensures the video is finalized and RAW images (if any) are saved.
* captureRawImage(): (Optional) Manually trigger a RAW capture during an active session (only if RAW is enabled). This allows capturing a RAW\_SENSOR frame on-demand while video is recording.
* **Internal Helpers:**
* Methods for setting up the camera device and outputs (e.g., openCamera(), configureSessionSurfaces()), building capture requests, handling thread initialization, etc., can be defined as private or internal.
* A callback interface (e.g., CameraRecorder.Callback) can be defined for events like errors or completion of recording, but this is optional and can also be handled via the returned SessionInfo.
* **Properties:**
* Camera device references (e.g., CameraDevice cameraDevice, CameraCaptureSession captureSession).
* Output objects: MediaRecorder mediaRecorder, ImageReader rawImageReader, and a reference to the Surface for the TextureView preview.
* Configuration flags (store whether video/raw were enabled for the current session).
* The background thread or coroutine context for camera operations.
* A SessionInfo sessionInfo object to track output file paths, timestamps, and any session metadata (this object can be constructed when starting a session).

The class should be designed for **modularity**, meaning each responsibility (preview setup, video recorder setup, raw capture logic, etc.) is handled in separate functions or logical blocks. This will make future extensions (like adding still JPEG capture, calibration triggers, or focus controls) easier to integrate without large modifications to the core logic.

## 2. Initialization Sequence and Camera Selection

**Camera initialization** involves obtaining the CameraManager and selecting the appropriate camera (likely the rear camera that supports the required capabilities). The steps are:

1. **Obtain CameraManager:** Use CameraManager manager = (CameraManager) context.getSystemService(CAMERA\_SERVICE) in Android. This gives access to the list of cameras on the device.
2. **Select Camera ID:** Iterate through manager.getCameraIdList() to find a camera that meets requirements. Specifically, choose a camera with at least **FULL** or **LEVEL\_3** hardware level and RAW capability. For example, check that CameraCharacteristics.REQUEST\_AVAILABLE\_CAPABILITIES contains CAPABILITIES\_RAW for that camera[[1]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=for%20%28String%20cameraId%20%3A%20manager,continue%3B). On Samsung S21/S22, the primary rear camera (usually ID "0") is likely a LEVEL\_3 device supporting RAW and high-resolution video. In most cases, you can select the first back-facing camera that supports RAW:
3. Use characteristics.get(LENS\_FACING) to find a back-facing camera.
4. Verify RAW support as mentioned.
5. (Also ensure the camera supports the needed output sizes: 4K video and RAW stream in parallel. The SCALER\_STREAM\_CONFIGURATION\_MAP in characteristics can be checked for supported output sizes and combinations.)
6. **Open the CameraDevice:** After selecting the camera ID, call manager.openCamera(cameraId, stateCallback, backgroundHandler/Executor). The **stateCallback** will handle onOpened(CameraDevice) and onDisconnected/onError events. On onOpened, save the CameraDevice reference (e.g., cameraDevice = openedDevice) and proceed to configure the session.  
   *Threading:* This call should be made on a background thread or with a dedicated executor so that camera initialization does not block the UI. (If using a HandlerThread, pass its Looper in openCamera; if using coroutines, ensure this runs on an IO/Background dispatcher.)
7. **Prepare Outputs on Initialization:** If the design allows, you might initialize certain outputs here. For instance, set up the TextureView listener (see Section 7) so that when its surface is ready, you can proceed with session creation. Also, if using a MediaRecorder, you can instantiate it here (but configure later), and similarly create an ImageReader for RAW (but actual session config happens in the next phase).

**Camera permission**: Ensure that by this point the app has camera permission; otherwise handle permission request before calling openCamera.

The camera selection logic ensures the chosen camera supports required streams. This is critical because not all cameras (e.g., front camera or ultra-wide) may support RAW or 4K video. By selecting the proper camera and verifying capabilities, we avoid runtime stream configuration failures.

## 3. Stream Configuration Logic (Preview, Video, RAW Outputs)

Once the camera is opened, the next step is to configure the output surfaces for the capture session. We need to set up streams for:

* **Preview (TextureView Surface):** The TextureView provides a SurfaceTexture which we convert to a Surface. We should set the SurfaceTexture’s default buffer size to match the chosen preview resolution. The preview resolution is usually chosen based on the display size or aspect ratio relative to the video size for efficiency. For example, if video is 4K (3840×2160, 16:9), we might use a 16:9 preview size as well (perhaps downscaled to fit the view). We can obtain supported preview sizes from StreamConfigurationMap.getOutputSizes(SurfaceTexture.class) and choose the one that fits the TextureView and matches the aspect ratio of the video[[2]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=,mTextureView.setAspectRatio%28mPreviewSize.getWidth%28%29%2C%20mPreviewSize.getHeight).
* **Video Recording (MediaRecorder Surface):** If video recording is enabled, configure the MediaRecorder (Section 5) and obtain its **Surface** via mediaRecorder.getSurface(). We need to decide on the video resolution and format: for 4K, that means 3840×2160 resolution, using an appropriate encoder. We should ensure this resolution is supported by the camera for recording:
* Query the StreamConfigurationMap.getOutputSizes(MediaRecorder.class) to see available recording sizes. On modern devices, 3840×2160 should be supported[[3]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=StreamConfigurationMap%20map%20%3D%20characteristics%20,class%29%2C%20width%2C%20height%2C%20mVideoSize). Choose 3840×2160 specifically for 4K unless the device does not list it (in which case pick the next highest resolution or use CamcorderProfile as a fallback).
* The MediaRecorder’s surface will be one of the session outputs. *Note:* Even if video is not enabled, one design choice is to always configure the session with the video surface (to avoid session reconfiguration when toggling recording on/off). However, including an unused surface has performance cost, so it's cleaner to only include it if recording is requested.
* **RAW Image Capture (ImageReader Surface):** If RAW capture is enabled, create an ImageReader with format ImageFormat.RAW\_SENSOR. The size should typically be the **maximum sensor resolution** for RAW to get full-quality Bayer data. We can get the largest supported RAW size from the camera characteristics: e.g., use map.getOutputSizes(ImageFormat.RAW\_SENSOR) and select the max dimensions[[4]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=Size%20largestJpeg%20%3D%20Collections,closed%20when%20all%20background%20tasks)[[5]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=new%20CompareSizesByArea,RAW_SENSOR%29%29%2C%20new%20CompareSizesByArea). For Samsung S21/S22, the main camera RAW resolution might be around 12MP or higher (e.g., 12MP sensor yields ~4000×3000). Use that full size for the ImageReader to capture maximum detail.
* Create the ImageReader with a capacity of a few images (e.g., ImageReader.newInstance(width, height, ImageFormat.RAW\_SENSOR, 2)), so it can hold at least 2 frames before dropping (this helps if we take RAW in quick succession or one is still being processed).
* Set an OnImageAvailableListener on this ImageReader to handle incoming RAW images (discussed in Section 6). This listener should operate on a background thread (we can use the same camera background thread or a separate one for image processing).
* **Combining Streams:** The Camera2 API allows multiple output targets to be active simultaneously in one capture session. We will supply a list of Surfaces to CameraDevice.createCaptureSession(...) that includes:
* Always: the preview Surface (for real-time display).
* If recording: the MediaRecorder Surface.
* If RAW: the RAW ImageReader’s Surface.

The camera device will attempt to configure all these streams at once. The hardware level (FULL/LEVEL\_3) guarantees some combinations. For instance, LEVEL\_3 devices **guarantee** RAW + preview + video concurrently in many cases. The exact supported combinations are device-specific, but since we filtered for a high-end device, the 3-stream combo should succeed. If a combination is unsupported (configuration failure), the app should handle the error gracefully (e.g., by falling back or informing the user).

* **Surface Lifecycle:** It’s important to keep these surfaces valid:
* The TextureView’s surface is tied to the UI lifecycle; ensure the TextureView is available (Section 7 covers waiting for it).
* The MediaRecorder surface is valid after MediaRecorder.prepare(). We must only call getSurface() after preparation (which we will do in session setup).
* The ImageReader surface is valid after creation; just ensure to close the ImageReader when done to free resources.

By planning the streams ahead, we ensure the CaptureSession is created with the right targets from the start. This avoids needing to tear down and recreate sessions for different modes mid-use. The logic essentially branches based on the recordVideo and captureRaw flags: for each true flag, include that output; if both are true, include both outputs alongside preview.

## 4. Session Start/Stop with CameraCaptureSession and CaptureRequest

**Starting the session:** Once surfaces are prepared, we create a camera capture session and start the capture requests:

* **Create CaptureSession:** Use cameraDevice.createCaptureSession(List<Surface> surfaces, StateCallback, handler). The surfaces list will be constructed as described in Section 3. Provide a CameraCaptureSession.StateCallback to handle completion:
* In onConfigured(CameraCaptureSession session): The camera is ready to use. Save the session (e.g., captureSession = session). Now prepare and send capture requests.
* In onConfigureFailed: Handle failure (e.g., log or throw an exception – likely not recoverable if it fails due to unsupported configuration).
* **Building Capture Requests:** We will typically use two types of requests:
* **Preview/Video Request (Repeating):** Create a CaptureRequest.Builder with appropriate template:
  + If recording video (or video+raw), use TEMPLATE\_RECORD for optimal steady frame rate[[6]](https://developer.android.com/media/camera/camera2/multiple-camera-streams-simultaneously#:~:text=%2F%2F%20You%20will%20use%20the,TEMPLATE_PREVIEW). This template prioritizes consistent frame timing which is important for video. It will also use continuous auto-focus/exposure suitable for recording.
  + If not recording (preview-only or preview+raw without video), TEMPLATE\_PREVIEW is appropriate, since it optimizes for low latency display[[6]](https://developer.android.com/media/camera/camera2/multiple-camera-streams-simultaneously#:~:text=%2F%2F%20You%20will%20use%20the,TEMPLATE_PREVIEW). (Either template would work for preview, but Preview template might reduce latency slightly.)
  + Always add the preview surface target to this request. If video recording is enabled, also add the MediaRecorder surface to the same request builder, so the frames are output to both targets in one go[[7]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mCameraDevice.createCaptureRequest%28CameraDevice.TEMPLATE_RECORD%29%3B%20List,addTarget%28recorderSurface). (This means the camera will feed both the preview and the encoder in parallel.)
  + Set any desired **default controls** on the request builder: e.g., builder.set(CaptureRequest.CONTROL\_MODE, CONTROL\_MODE\_AUTO) to let auto-exposure/auto-focus run[[8]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=,the%20camera%20preview%20size%20is). Typically, for recording and preview, we rely on continuous autofocus (CONTROL\_AF\_MODE\_CONTINUOUS\_VIDEO) and auto-exposure.
  + Once built, call session.setRepeatingRequest(builder.build(), captureCallback, backgroundHandler). Here, captureCallback can usually be null or minimal for preview; we don’t need per-frame feedback for preview in this scenario, so often null is fine. We do supply the background handler so that the request processing is on a background thread[[9]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=setUpCaptureRequestBuilder,thread.start%28%29%3B%20mPreviewSession.setRepeatingRequest%28mPreviewBuilder.build%28%29%2C%20null%2C%20mBackgroundHandler).
* **RAW Capture Request (Single):** When we need to capture a RAW image (either at session start or on demand), we create a one-time capture:
  + Use TEMPLATE\_STILL\_CAPTURE for the capture request to get a high-quality still frame (this may use slower but higher-quality settings)[[6]](https://developer.android.com/media/camera/camera2/multiple-camera-streams-simultaneously#:~:text=%2F%2F%20You%20will%20use%20the,TEMPLATE_PREVIEW). This is ideal for RAW to get a well-exposed, sharp image. If video is simultaneously running, using STILL template will momentarily use settings for a photo (which might, for example, use a shorter exposure if flash or similar – but since we likely won’t use flash in a parallel capture scenario, it should be fine).
  + Add the RAW ImageReader’s surface as the target. We can also add the preview surface here if we want the capture to also update the preview (though not strictly necessary; the preview is anyway running). It’s common to just target the RAW surface for simplicity.
  + (Optional) If we want the RAW capture to be synchronized with a specific video frame, we could add both RAW and one of the active surfaces to the same capture request. For instance, adding RAW + preview surfaces to a single capture() call ensures the preview (and video, if it’s part of repeating) doesn’t get out of step. However, it’s usually fine to capture RAW alone; the camera device will handle coordinating sensor reads.
  + Submit this request with session.capture(rawRequest, captureCallback, backgroundHandler). Provide a CaptureCallback to retrieve the TotalCaptureResult for this capture. This result carries metadata (like exposure time, sensor settings) needed for saving the DNG file.
* **Starting Video Recording:** If video is enabled, the MediaRecorder should start receiving frames as soon as the repeating request with its surface is active. We actually need to call mediaRecorder.start() to begin recording to file. A typical flow is:
* Configure and prepare MediaRecorder (before session creation or in the onConfigured step).
* Once the capture session is configured and the repeating request has started, call mediaRecorder.start() to begin writing the video file. In our design, we might start the repeating request and immediately start the recorder. (If we included the recorder surface in the request, frames were being fed to the encoder; start() will actually begin the encoding/writing process.)
* At this point, the video is being recorded to the output file.
* **SessionInfo and Timestamps:** When the session successfully starts, update the SessionInfo object with things like start time (System.currentTimeMillis) and confirmation of which outputs are active. For instance, record the file path of the video and a list or counter for RAW images that will be saved (maybe start an index at 0 if multiple RAWs will be taken).

**Stopping the session:** Cleanly shutting down is crucial to avoid resource leaks or corrupt files:

* **Stop MediaRecorder:** If a video was recording, call mediaRecorder.stop() to finalize the file. This writes the trailer/moov atoms for MP4. Then call mediaRecorder.reset() or release() to free the encoder[[10]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=%2F%2F%20UI%20mIsRecordingVideo%20%3D%20false%3B,%2B%20getVideoFile%28activity)[[11]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mMediaRecorder,startPreview). (After stop(), the video file is complete and can be accessed via its path in SessionInfo.)
* **Finish/Flush Pending RAW captures:** If a RAW capture was in progress or the ImageReader has an image not yet saved, ensure those are handled. In practice, by the time we stop, any triggered RAW capture should have been processed (we handle saving in Section 6’s logic). If needed, one can call captureSession.abortCaptures() to cancel any in-flight captures, but usually not required unless we had a burst ongoing. We should close the ImageReader after use (but only after all images are acquired and saved).
* **Close CameraCaptureSession:** Call captureSession.close(). This will stop the preview. It’s good to do this after stopping the MediaRecorder to ensure no further frames are sent. Also close the CameraDevice via cameraDevice.close() after the session is closed or in its onClosed callback.
* **Release Surfaces:**
* TextureView’s surface will be released when the camera device is closed (the SurfaceTexture can remain for UI or be freed by calling texture.release() if you want).
* The ImageReader should be closed (rawImageReader.close()), which will free its buffers.
* The MediaRecorder should be released (mediaRecorder.release()) if not reusing it for another session immediately (especially if we called reset, we might reuse the object, but releasing is safer if we recreate for each session).
* **Update SessionInfo:** Mark the session as stopped, note the stop time, and ensure the output file paths are recorded. If multiple RAW images were captured, list their filenames in SessionInfo so that the session’s data is fully described.

All the stop logic should ideally happen on the background thread or by ensuring the camera operations (stop captures, closing device) are off the UI thread, since these can stall for a few moments (especially mediaRecorder.stop() can block until the file is finalized).

By structuring start/stop carefully, we ensure a robust lifecycle for the camera session, avoiding issues like camera in use by others, or leaked resources. This also sets the stage for potentially restarting sessions back-to-back if needed (each startSession should pair with a stopSession before starting a new one, or reuse the same open camera if desirable, though simplest is open/close per session unless quick restarts are needed).

## 5. MediaRecorder Setup for 4K Video (No Audio)

Setting up the MediaRecorder for 4K video is a crucial step for the video recording functionality. We will use H.264 encoding (with an option to extend to H.265/HEVC if needed later) and we omit audio for simplicity. Key configuration steps include:

* **Create and Configure MediaRecorder:** Typically done before starting the capture session (e.g., right after camera open, or in startSession before creating the session). In Kotlin/Java:
* mediaRecorder = new MediaRecorder() (if not already created earlier).
* **Audio Source:** *No audio* is needed, so we do not call setAudioSource. (By default, not setting an audio source means no audio track.) If an audio track were required in future, we’d use MediaRecorder.AudioSource.MIC etc., but for now skip this.
* **Video Source:** Use MediaRecorder.VideoSource.SURFACE[[12]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mMediaRecorder,AAC). This indicates we’ll get the video input from a Surface (provided to camera). This call is necessary to put MediaRecorder into the correct state.
* **Output Format:** mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG\_4). MP4 is a common container for H.264/H.265.
* **Video Encoder:** mediaRecorder.setVideoEncoder(MediaRecorder.VideoEncoder.H264) for H.264 AVC codec[[13]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mMediaRecorder,AAC). (Samsung devices also support HEVC/H.265 via VideoEncoder.HEVC; we could allow selecting this as an option, but default H.264 for wider compatibility.)
* **Video Resolution:** mediaRecorder.setVideoSize(3840, 2160) for 4K UHD[[14]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mMediaRecorder,get%28rotation). Make sure this matches one of the supported sizes. If 4K is not supported by a particular device (unlikely for S21/S22, since they do support 4K), we would choose a supported size (maybe 1080p). We already picked this size from the camera config map.
* **Frame Rate:** mediaRecorder.setVideoFrameRate(30). 30fps is standard for 4K on most phones. (If needed, we can make this dynamic or 60fps if supported, but typically 30 to ensure RAW capture concurrency doesn’t overload the sensor.)
* **Bitrate:** Set a high bitrate for quality. For example, mediaRecorder.setVideoEncodingBitRate(10000000) (10 Mb/s) or higher for 4K[[15]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mMediaRecorder,get%28rotation). We might choose ~20Mb/s for better quality at 4K. This can be tuned or derived from CamcorderProfile (CamcorderProfile for QUALITY\_2160P usually provides a recommended bitrate).
* **Orientation Hint:** Set the correct rotation so the video is not rotated wrongly. Use the device rotation to compute orientation: e.g.,
* int rotation = activity.getWindowManager().getDefaultDisplay().getRotation();  
  int orientationHint = ORIENTATIONS.get(rotation);  
  mediaRecorder.setOrientationHint(orientationHint);
* This ensures the recorded video is tagged with the display orientation (90, 180, 270 degrees if needed). For Samsung phones, this is important if recording in portrait, so that players can rotate it correctly. (ORIENTATIONS is typically a mapping of Surface.Rotation to degrees.)
* **Output File:** Provide a file path where the video will be saved: e.g., mediaRecorder.setOutputFile(sessionInfo.videoFilePath). We should generate a unique file name (like using timestamp or an incrementing index, e.g., "Session\_"+ sessionId + ".mp4"). On S21/S22 (Android 11/12), it’s safe to use app-specific storage (context.getExternalFilesDir(Environment.DIRECTORY\_MOVIES) or similar) so we don’t need external storage permission. Ensure the directory exists and have write access.
* **Prepare MediaRecorder:** Call mediaRecorder.prepare(). This initializes the encoder and prepares the output file. After prepare(), the mediaRecorder.getSurface() becomes available to use in the camera session. We must call prepare **before** creating the camera capture session (because we need the Surface). The official sample does this right before session creation[[16]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=setUpMediaRecorder%28%29%3B%20SurfaceTexture%20texture%20%3D%20mTextureView,getHeight)[[7]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mCameraDevice.createCaptureRequest%28CameraDevice.TEMPLATE_RECORD%29%3B%20List,addTarget%28recorderSurface):
* After prepare, retrieve the Surface: Surface recorderSurface = mediaRecorder.getSurface(). We will include this in the session’s surface list.
* **No Audio Consideration:** By not calling setAudioSource and not setting an audio encoder, the output file will have no audio track. This simplifies things. (MediaRecorder in Android is fine with video-only recording as long as we skip audio config.)
* **Starting and Stopping Recording:**
* After the capture session is configured and repeating requests are running, start the recording by calling mediaRecorder.start()[[17]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mIsRecordingVideo%20%3D%20true%3B%20%2F%2F%20Start,). This begins writing the video file with incoming frames.
* On stop, as mentioned, use mediaRecorder.stop() then release(). Stopping properly flushes the encoder’s buffers and finalizes the file. Always guard stop() in try-catch because if something went wrong (e.g., no frames or too short duration), it can throw an exception.

By following these steps, we ensure a smooth video recording setup. The MediaRecorder is effectively acting as a consumer of the camera frames. Using the **Camera2 + MediaRecorder** path leverages hardware encoders for efficiency. One must ensure to handle errors: e.g., if MediaRecorder.prepare() fails (perhaps due to wrong settings or missing permissions for file), handle that by logging or informing the user. Also, if recording is started and an error occurs (MediaRecorder can invoke OnErrorListener), those should be captured (we can set mediaRecorder.setOnErrorListener to handle catastrophic failures).

In summary, this setup will yield a 4K H.264 .mp4 file with no audio for each session when video is enabled.

## 6. RAW Capture Logic with ImageReader and DngCreator

When RAW capture is enabled, we need to handle the output of the ImageReader that provides RAW sensor images and then save those to DNG files. Key components in this process:

* **ImageReader Listener:** After creating the ImageReader for RAW\_SENSOR format, set an OnImageAvailableListener. This will be called when a RAW image frame is captured and ready. We should set this up on a background thread (pass a Handler from the camera thread, or if using coroutines, perhaps use a separate single-thread executor for image I/O). For example:
* rawImageReader.setOnImageAvailableListener(reader -> {  
   Image image = reader.acquireNextImage();  
   if (image != null) {  
   handleRawImage(image);  
   }  
  }, backgroundHandler);
* where handleRawImage(Image) is a function we implement to process and save the raw data.
* **Capture Trigger:** How RAW images are captured depends on use-case:
* If **both video and RAW** are enabled (parallel mode), we might capture a RAW image at certain trigger points (e.g., when the session starts, or when the user presses a capture button during recording, via the captureRawImage() method in our API). It’s generally not feasible to capture RAW at video frame-rate continuously (due to large file size and sensor throughput), so triggers are better. We use cameraCaptureSession.capture(rawRequest, callback, backgroundHandler) as described in Section 4 to capture each RAW frame on demand.
* If **RAW-only mode** (video disabled, RAW enabled), we can either capture a single RAW photo (like a still capture) or possibly a sequence of RAW images if needed. For a basic implementation, capturing one RAW when the user triggers it (like pressing a shutter) is typical. If a continuous sequence is required (unlikely due to storage), we’d loop captures with some interval.
* **Using DngCreator:** Android’s DngCreator class helps convert the Image from RAW\_SENSOR into a .dng file. The process to save a RAW image is:
* Obtain CameraCharacteristics (we have it from camera selection) and the CaptureResult associated with that image. In our CaptureCallback for the RAW capture, when onCaptureCompleted fires with a TotalCaptureResult, save that result (the metadata) for use. For example, in the capture callback: if (result.getRequest().getTargetSurfaces().contains(rawImageReader.getSurface())) { lastRawCaptureResult = result; }. We can match by a tag or just store the latest capture result for RAW.
* In handleRawImage(Image image):
  + Ensure we have the TotalCaptureResult corresponding to this image. (Camera2 ensures that a capture which outputs an Image to an ImageReader will have a TotalCaptureResult delivered before the OnImageAvailable for that image, typically. We might store the result in a variable that the image listener can access, or pass via a small thread-safe queue.)
  + Create a DngCreator dngCreator = new DngCreator(cameraCharacteristics, totalCaptureResult). This binds the metadata (like color correction, exposure, etc.) with the raw pixel data[[18]](https://stackoverflow.com/questions/57126430/taking-a-dng-picture-using-the-camera2-api#:~:text=if%20%28mImage.format%20%3D%3D%20ImageFormat.RAW_SENSOR%29%20,finally)[[19]](https://stackoverflow.com/questions/57126430/taking-a-dng-picture-using-the-camera2-api#:~:text=,be%20saved%2C%20for%20example).
  + Prepare an output file. We should generate a filename for the DNG. Possibly use the SessionInfo’s session ID or timestamp plus an index. For example, Session123\_Raw1.dng, Raw2.dng for subsequent images. The path can be in Pictures/ or app-specific storage (e.g., ExternalFilesDir(DIRECTORY\_PICTURES)).
  + Open a FileOutputStream to that file.
  + Call dngCreator.writeImage(outputStream, image)[[18]](https://stackoverflow.com/questions/57126430/taking-a-dng-picture-using-the-camera2-api#:~:text=if%20%28mImage.format%20%3D%3D%20ImageFormat.RAW_SENSOR%29%20,finally). This will write the DNG format file with all the necessary tags.  
    *Note:* DngCreator will use the image.getPlanes()[0].getBuffer() and other planes internally, no need for manual byte buffer copying if using writeImage.
  + Close the outputStream and then call image.close() to free the ImageReader buffer[[20]](https://stackoverflow.com/questions/57126430/taking-a-dng-picture-using-the-camera2-api#:~:text=try%20,mImage.close%28%29%20closeOutput%28output%29).
  + Optionally, call dngCreator.close() (not always necessary to close explicitly, but it’s good practice to allow GC).
* Wrap the above in try/catch for IO exceptions. Also handle cases where image might be null or if totalCaptureResult is not available (shouldn’t happen in normal operation if we coordinate properly).
* **Threading and performance:** Writing a DNG (especially at ~12-16MP raw) can be slow (tens of milliseconds to a few hundred). This must be done on a background thread to not stutter the UI. If using the camera’s single background thread, note that writing to disk could block other camera operations if done on the same thread. To avoid this, you can offload DNG saving to a separate IO-dedicated coroutine or thread:
* For instance, use CoroutineScope(Dispatchers.IO).launch { dngCreator.writeImage(...) } so that the camera background thread is free to continue. Or maintain an executor specifically for file I/O.
* The Google sample uses a queue and a reference-counted closeable to ensure the ImageReader isn’t closed until save is done[[21]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=synchronized%20%28mCameraStateLock%29%20,5%29%29%3B) – this is an advanced detail. Simpler approach: increase the ImageReader capacity and ensure each image is closed after saving.
* **Memory management:** RAW images are large. Closing the Image (and optionally using only one image at a time by acquireLatestImage) is important to avoid memory buildup. We use acquireNextImage() or acquireLatestImage() carefully:
* If capturing single images occasionally, acquireNextImage() is fine (take them in order).
* Always close the image after use (inside finally block).
* If not closed, the ImageReader buffer queue will fill (maxImages limit) and block the camera.
* **DNG Metadata:** The DngCreator automatically includes metadata from the CaptureResult, such as exposure time, sensor sensitivity (ISO), etc., into the DNG file. It will also embed the ColorCorrection matrix and other info from CameraCharacteristics. If needed, one can also set the orientation tag via dngCreator.setOrientation() if the device rotation should be recorded (though many RAW shooters handle orientation separately; since RAW is raw data, one might not rotate it and instead interpret orientation for viewing). For completeness, if the app knows the device orientation when the RAW was captured, you could do:
* dngCreator.setOrientation(cameraCharacteristics.get(CameraCharacteristics.SENSOR\_ORIENTATION));
* or use the device rotation to set a tag (the DNG spec orientation tag usage can be optional).
* **Integration with SessionInfo:** Each time a RAW image is saved, record its filename or identifier in the SessionInfo. For example, add the path to a list like sessionInfo.rawFiles.add(path). This way, after the session, we know all files produced. If only one RAW is captured per session (raw-only mode), you can store a single file path. If multiple, an array or count.

By implementing this RAW capture pipeline, the module will be able to produce .dng files that can be opened in tools like Adobe Lightroom or analyzed for sensor data. This satisfies the requirement of capturing RAW\_SENSOR (Bayer) images. The approach uses best practices from Camera2: using the DngCreator utility rather than writing our own DNG, and capturing metadata from the actual frame[[18]](https://stackoverflow.com/questions/57126430/taking-a-dng-picture-using-the-camera2-api#:~:text=if%20%28mImage.format%20%3D%3D%20ImageFormat.RAW_SENSOR%29%20,finally).

One must ensure that the camera is **in focus** and exposure is stable when capturing RAW. The preview (or video) auto-focus and auto-exposure are running continuously; if a RAW capture is triggered, it will use whatever current focus/exposure settings are at that moment. For critical quality, the app might consider locking AE/AF before capturing RAW (this is what a typical still capture would do). For a simpler implementation, we may skip explicit AE/AF lock, but it's a point for future improvement or testing (particularly if RAW images turn out blurry or improperly exposed due to ongoing adjustments).

## 7. TextureView Binding for Preview Surface

For live viewfinder display, we use a TextureView in the UI to show the camera preview frames. Key steps to bind the TextureView to the camera preview:

* **TextureView Setup:** In the UI (activity/fragment layout), a TextureView is placed for the preview. We recommend using an AutoFitTextureView (as seen in Google samples) or managing the aspect ratio to prevent the preview from appearing stretched. The aspect ratio should match the chosen preview size (which we aligned with video aspect ratio, likely 16:9 for 4K).
* **SurfaceTextureListener:** Because the camera needs a **Surface** from the TextureView, we must wait until the TextureView is ready. Use textureView.setSurfaceTextureListener(surfaceTextureListener). The listener callbacks of interest:
* onSurfaceTextureAvailable(SurfaceTexture surface, int width, int height): Called when the TextureView is first ready. At this point, we can obtain the SurfaceTexture.
  + We should configure the size of the SurfaceTexture buffer to our desired preview size: surface.setDefaultBufferSize(previewWidth, previewHeight). This ensures the camera output frames are at the correct resolution for the TextureView[[16]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=setUpMediaRecorder%28%29%3B%20SurfaceTexture%20texture%20%3D%20mTextureView,getHeight)[[22]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=texture).
  + Then create a Surface: Surface previewSurface = new Surface(surface). This is what we will use as the camera output target.
  + If the camera is already opened and we have the capture session config ready, we can now proceed to create the capture session with this previewSurface. In practice, you might start openCamera() in onSurfaceTextureAvailable to ensure the surface is ready when you configure the session.
  + If we called initialize() earlier with the TextureView, our code might have been waiting for this callback. We can now call our internal startCaptureSession() logic since surfaces are set.
* onSurfaceTextureSizeChanged(...): We may want to adjust the view transform matrix here (for example, handle rotation or scaling – the Google sample uses configureTransform() for rotation correction[[23]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=public%20void%20onOrientationChanged%28int%20orientation%29%20,)[[24]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=private%20void%20configureTransform,getHeight)).
* onSurfaceTextureDestroyed(...): Return true to let it release, and we should handle closing the camera if the UI is going away. Typically, when the TextureView is destroyed (e.g., activity stops), we should stop the session (to avoid the camera feeding into a surface that no longer exists).
* onSurfaceTextureUpdated(...): Not crucial for our logic; it’s called every frame for preview updates if needed for additional processing.
* **Binding in Code:** If our CameraRecorder.initialize(textureView) is called, inside it we can do:
* if (textureView.isAvailable()) {  
   // SurfaceTexture already available (perhaps view was already laid out)  
   SurfaceTexture st = textureView.getSurfaceTexture();  
   st.setDefaultBufferSize(previewWidth, previewHeight);  
   Surface previewSurface = new Surface(st);  
   // proceed to open camera and configure session with previewSurface  
  } else {  
   textureView.setSurfaceTextureListener(surfaceListener);  
   // In onSurfaceTextureAvailable, do the above.  
  }
* This ensures we don’t miss the event if the TextureView became available before we set the listener (common when returning from background, etc.). The Google sample demonstrates checking isAvailable() in onResume[[25]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=%2F%2F%20the%20SurfaceTextureListener%29,canDetectOrientation)[[26]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=configureTransform%28mTextureView.getWidth%28%29%2C%20mTextureView.getHeight%28%29%29%3B%20%7D%20else%20,mOrientationListener.enable%28%29%3B%20%7D).
* **Ensuring Correct Orientation:** The camera sensor orientation (90 or 270 degrees usually for portrait sensors) might cause the preview to appear rotated on the screen. On a TextureView, we can correct this by applying a transform matrix if needed. The sample method configureTransform() calculates a matrix to rotate/scale the preview to fit the view dimensions[[27]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=int%20rotation%20%3D%20activity,bufferRect.centerY)[[28]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=if%20%28Surface.ROTATION_90%20%3D%3D%20rotation%20,2%29%2C%20centerX%2C%20centerY). In our guide, we can mention:
* For simplicity, if we lock the app to portrait or handle orientation differently, we might skip this. But ideally, handle it: e.g., if the device is rotated to landscape, the preview needs a 90° rotation. The matrix uses textureView.setTransform(matrix) to achieve that.
* For S21/S22, the main camera likely has a sensor orientation of 90° (typical for landscape natural orientation). If the app UI is in portrait, we need to rotate preview by 90. We should thus utilize the device rotation from WindowManager and set transform accordingly (as in the sample code).
* **Preview Frame Rate:** The preview will run as fast as the capture request allows (usually 30fps when recording 4K, or higher if the template is preview and the device supports it). The TextureView should handle that on the UI thread. If the UI thread is busy, preview might stutter, so keep heavy operations off the UI.

By correctly binding the TextureView, the user will see a **live RGB preview** on the screen, which meets the requirement. The preview is independent of whether we are recording or capturing raw; it’s always on to help frame the shot.

Testing hint: ensure that the preview is not overly stretched or cropped – the aspect ratio management is key. If using an AutoFitTextureView (a custom TextureView that adjusts its size to a given aspect ratio), set it to the aspect of previewSize[[2]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=,mTextureView.setAspectRatio%28mPreviewSize.getWidth%28%29%2C%20mPreviewSize.getHeight)[[29]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mPreviewSize%20%3D%20chooseOptimalSize%28map,mTextureView.setAspectRatio%28mPreviewSize.getWidth%28%29%2C%20mPreviewSize.getHeight). For example, if preview is 16:9, the view should be 16:9 to avoid black bars.

## 8. Coroutine/Threading Model for Session Lifecycle

Managing threads or coroutines is vital for a responsive and crash-free camera module. The Camera2 API is asynchronous, but many callbacks (camera open, session configured, image available) can be directed to a background thread. Here we design a threading model:

* **Background Thread (HandlerThread) Approach:** A common pattern is to create a HandlerThread for camera operations:
* Start a HandlerThread ("CameraThread") in initialize() or on camera start. Create a Handler backgroundHandler = new Handler(cameraThread.getLooper()). All camera callbacks (openCamera, session state, image reader) can use this handler.
* This means CameraDevice.openCamera(..., backgroundHandler) and createCaptureSession(..., backgroundHandler) will execute callbacks on that thread. Also ImageReader.setOnImageAvailableListener(..., backgroundHandler).
* This single thread serializes all camera events (which is often fine, since camera operations happen sequentially). It prevents blocking the UI thread with heavy operations.
* In stopSession(), we should quit this thread (e.g., handlerThread.quitSafely() and join) to clean up.
* **Kotlin Coroutines Approach:** Alternatively, and especially if the rest of the project uses coroutines, we can encapsulate camera operations in a coroutine-friendly way:
* Use a single-threaded context for camera, e.g., private val cameraDispatcher = Executors.newSingleThreadExecutor().asCoroutineDispatcher(). This gives a coroutine dispatcher confined to one background thread (similar effect to HandlerThread).
* Run blocking camera calls on this dispatcher: e.g.,
* withContext(cameraDispatcher) {  
   cameraDevice = openCameraSuspend(cameraManager, cameraId)  
   // configure session, etc.  
  }
* where openCameraSuspend could be a suspend function using suspendCancellableCoroutine to open the camera and wait for onOpened callback.
* For simpler implementation, one might still rely on the callback style but just initiate them on this background context.
* Use coroutines for higher-level sequencing: for example, startSession() could be a suspend function that does: open camera -> create session -> start recording, all inside a coroutine on cameraDispatcher. This linearizes the workflow nicely without nested callbacks.
* The ImageReader.onImageAvailableListener could launch a coroutine on an IO dispatcher to save the image, rather than doing it on the callback thread directly.
* **UI Thread Interactions:** Keep UI updates (like updating a record button state, or showing a Toast) on the main thread. For instance, after starting recording or when saving is complete, if we need to notify the user, we must switch to Main thread (using withContext(Dispatchers.Main) { ... } or posting to a Handler on Looper.getMainLooper()). The camera module can expose LiveData or callbacks for the UI layer to observe, rather than directly doing UI ops internally.
* **Synchronization and Locks:** It’s wise to use some locking or flags to avoid race conditions:
* For example, use a Semaphore cameraLock = new Semaphore(1) to prevent opening the camera while it’s closing (the Google sample does this to avoid fast reuse issues[[30]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=if%20%28%21mCameraOpenCloseLock.tryAcquire%282500%2C%20TimeUnit.MILLISECONDS%29%29%20,SCALER_STREAM_CONFIGURATION_MAP)[[31]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=try%20,release%28%29%3B%20mMediaRecorder%20%3D%20null)). Acquire it before open and release after close. Similarly, ensure startSession() is not called twice concurrently.
* If using coroutines, a simple mutex or actor model could ensure only one session runs at a time.
* **Thread Cleanup:** When the host (Activity/Fragment) pauses or the session stops, ensure to cancel or shut down threads:
* For HandlerThread: quitSafely and nullify the handler.
* For coroutine dispatcher: call cameraDispatcher.close() or shut down the Executor service.
* Cancel any ongoing coroutines if the activity is stopping (to avoid callbacks to a destroyed UI).

By following these threading guidelines, the camera operations (which can involve waits for hardware and IO) won’t freeze the UI. The use of a single background thread (or single-threaded coroutine dispatcher) is a best-practice for Camera2 to avoid concurrency issues. The camera APIs themselves are not thread-safe across multiple threads calling different methods, so funneling through one thread is simplest.

**Example:** The preview update is on the camera thread and does not block UI; the Image saving is on an IO thread so it doesn’t block the camera thread more than necessary. This separation ensures smooth preview at 30fps while maybe writing a large DNG file concurrently (the frame pipeline might stall a bit if RAW capture is in progress, but the preview should continue since it’s a separate stream).

In summary, the coroutine/threading model should make the session lifecycle (open -> start -> running -> stop -> close) safe and predictable. Document these thread boundaries clearly in code comments so future maintainers know which context things run in.

## 9. File Output Management and SessionInfo Integration

The SessionInfo is intended to track details about each recording session, including file outputs. Designing the file management with SessionInfo ensures data from different sessions don’t mix and are easy to reference. Consider the following:

* **SessionInfo Structure:** This could be a simple data class with fields such as:
* sessionId or timestamp.
* videoFilePath (String, if video was recorded; null if not).
* rawFilePaths (List<String> for RAW DNG files; empty if none).
* startTime and endTime (timestamps) of the session.
* Maybe flags or settings used (e.g., videoEnabled, rawEnabled booleans for record).
* Possibly metadata like which camera was used, or any errors that occurred.
* **File Naming Scheme:** To organize outputs, use a consistent naming convention. For example:
* Base name with session identifier: if using timestamp, Session\_<yyyyMMdd\_HHmmss>\_... etc.
* For video: Session\_<id>.mp4
* For raw: Session\_<id>\_RAW\_<index>.dng (index if multiple raw images).
* Alternatively, create a dedicated subdirectory per session (especially if multiple files). E.g., .../Session\_<id>/video.mp4 and .../Session\_<id>/raw\_1.dng, raw\_2.dng, etc. This makes it easy to manage or delete a whole session’s files together.
* **File Paths and Permissions:** As noted, using app-specific storage (e.g., Context.getExternalFilesDir) is convenient. On modern Android, this doesn’t require runtime WRITE permissions and the files are accessible via USB for debugging. Ensure to pick the appropriate directory:
* Videos could go in .../Movies/ or .../DCIM/ under the app folder.
* RAW images could go in .../Pictures/ or the same folder.  
  SessionInfo can hold the absolute paths or Uris of these files.
* **Creating Files:** At startSession, generate the file names:
* For instance, when video is enabled:
* File videoFile = new File(context.getExternalFilesDir(Environment.DIRECTORY\_MOVIES),  
   baseName + ".mp4");  
  sessionInfo.videoFilePath = videoFile.getAbsolutePath();  
  mediaRecorder.setOutputFile(sessionInfo.videoFilePath);
* For raw, perhaps do not create the file upfront, but prepare a base path. Each time captureRawImage() is called or at session end, create a new File:
* File rawFile = new File(context.getExternalFilesDir(Environment.DIRECTORY\_PICTURES),  
   baseName + "\_RAW\_" + rawCount + ".dng");
* where rawCount increments for each capture (starting at 1). Add each to sessionInfo.rawFilePaths.
* **After Recording Ends:** The SessionInfo now contains all the output references. We should ensure the data is flushed to disk:
* Video file is finalized when mediaRecorder.stop() returns. We might want to scan it if we need it to appear in gallery (MediaStore), but if just internal use, not necessary.
* The DNG files are written as we go. After each DNG save, we could optionally update SessionInfo immediately or at the end.
* If any file fails to save (e.g., IO error), we should handle that: remove it from SessionInfo or mark a flag that an error occurred.
* **Integration:** If the SessionInfo is part of a larger system (maybe they keep logs or a database of sessions), ensure to pass this SessionInfo object back to the caller or to whatever system stores it. For example, startSession() could return the SessionInfo, or it might be stored internally and accessible via getCurrentSessionInfo(). Once stopSession() completes, SessionInfo can be considered finalized and can be archived or displayed.
* **Future Extension (Still capture or calibration files):** The file management should be flexible to add more types. For instance, if a “still JPEG capture” was added in future, SessionInfo could have a field for photoFilePath. Or if a calibration produces a data file, that could be included. By having a unique session folder or ID, grouping these is easier.
* **Cleanup:** Optionally, implement a cleanup strategy for old sessions (not strictly required in this guide, but practically useful). E.g., if storage is a concern, provide a way to delete old session files either manually or automatically after upload, etc.

In essence, SessionInfo binds together the toggles and the outputs of a recording session. This modular approach means outside code can use SessionInfo to get all relevant info (for example, to upload files or to display to user “Video saved at X, 3 RAW images saved at Y”). It also enforces that file creation and naming is handled in one place, reducing mistakes like naming collisions.

Example snippet integrating SessionInfo:

SessionInfo sessionInfo = new SessionInfo(sessionId);  
sessionInfo.videoEnabled = recordVideo;  
sessionInfo.rawEnabled = captureRaw;  
sessionInfo.startTime = System.currentTimeMillis();  
if (recordVideo) {  
 sessionInfo.videoFilePath = videoFile.getAbsolutePath();  
}  
...  
// After capturing raw  
sessionInfo.rawFilePaths.add(rawFilePath);  
...  
sessionInfo.endTime = System.currentTimeMillis();

This object can then be logged or passed around. Using it internally also helps track state (like knowing whether to expect raw captures, etc., by checking the flags in sessionInfo instead of separate boolean vars).

## 10. Manual Test Plan

To ensure the CameraRecorder module works as expected on the Samsung S21/S22 (and similar Camera2 FULL/LEVEL\_3 devices), perform the following **manual tests** covering all features and combinations:

1. **Baseline Preview Test:** Initialize the CameraRecorder with preview only (both video and raw toggles off, if the design allows, or simply do not call startSession yet). Ensure that:
2. The TextureView displays a live camera feed.
3. The preview is smooth (30fps) and correctly oriented. Rotate the device to confirm the preview rotates/fits (if orientation changes are supported).
4. No visible distortion (aspect ratio correct).
5. **Video-only Recording Test:** Start a session with recordVideo=true, captureRaw=false. Verify:
6. The preview remains active during recording.
7. Video recording starts (perhaps indicate via UI). Let it run for a short period (e.g., 10 seconds).
8. Stop the session. Confirm that a video file is saved (check SessionInfo for the path).
9. Play back the video file on the device: verify 4K resolution, smooth playback, and that there is **no audio track** (the video should be silent). Also check orientation: if you recorded in portrait, does it play rotated correctly (Orientation hint applied)?
10. Check file size roughly corresponds to the bitrate and duration (to ensure bitrate setting took effect).
11. Repeat test for both rear and (if supported by implementation) front camera if applicable – but mainly our focus is the main rear camera.
12. **RAW-only Capture Test:** Start a session with recordVideo=false, captureRaw=true. This mode is essentially for capturing RAW images:
13. Upon starting, if the design automatically captures a RAW frame, ensure a RAW image is saved. If not automatic, use the provided API (e.g., press a capture button that calls captureRawImage()).
14. Verify the preview is still visible (it should be, even if not recording video).
15. If one RAW image is captured, check SessionInfo for the .dng file path. Using a PC or a mobile app that can open DNG (like Adobe Lightroom Mobile or Snapseed), open the DNG file:
    * Verify it contains a valid image (not corrupted, correct size). The image might appear dark or flat (RAW images often do before editing), but it should not be pure black or noisy garbage – that would indicate a failure in capture or saving.
    * Check metadata in the DNG (if possible, using a tool): it should have sensible values for exposure, ISO, etc., matching the scene (this confirms CaptureResult was applied).
16. If the design allows multiple RAW captures in one session, trigger another capture and verify a second file is saved and is valid.
17. Test edge cases: capture RAW back-to-back quickly (if supported) to see if the app can handle it or if it needs a short delay. Also ensure the app doesn’t freeze during the saving (the preview might stutter briefly during disk write, but it should recover).
18. **Concurrent Video + RAW Test:** This is the key scenario (both toggles on):
19. Start a session with recordVideo=true, captureRaw=true. The preview should start and video recording begins.
20. While recording, trigger a RAW capture (via UI action or perhaps automatically at start).
    * Confirm that the video continues recording while the RAW is being captured. There might be a slight hiccup in preview or a dropped frame in video when taking a RAW (depending on device), but the recording should not stop or crash.
21. You might hear a camera shutter sound if the system plays one on capture – that’s normal when taking stills.
22. After capturing a RAW, maybe capture another one a few seconds later, to test multiple RAWs during one video.
23. Stop the recording after, say, 10-15 seconds.
24. Verify outputs: one MP4 video file and multiple DNG files (count matches how many were triggered).
25. Check the video file: it should have the full duration and play fine. Scrub through it near the times RAW was captured – see if any major disruption (a minor pause or exposure change might happen for one frame, but it should be generally fine). The expectation on a high-end device is minimal disturbance.
26. Check the DNG files: open them to ensure they are valid. If possible, correlate the timing: e.g., a RAW captured mid-video might show the scene that was being recorded (perhaps even motion if something in scene moved).
27. This test is critical on both S21 and S22: these devices should handle this, but watch for any device-specific issues (for example, some devices might not allow RAW at 60fps video, etc. If S21/S22 support 60fps 4K, perhaps our default 30fps is safe).
28. **Toggle Behavior Test:** Test the transitions and edge conditions:
29. Start video-only, then stop, then start a new session with video+raw, etc., in one app run. Ensure each session’s outputs are correct and previous session’s state does not leak (the module should be re-usable for multiple sessions sequentially).
30. If the design allowed changing mode without app restart (like toggling a switch and calling startSession again), test doing so (though typically you stop then start new session).
31. Test if the app handles “early stop”: e.g., start recording and stop after 1 second. Video file should still be saved and playable (short clip). Similarly, trigger a RAW and stop immediately – ensure the RAW still saved.
32. **Resource Cleanup Test:** After stopping a session, attempt to start another immediately. This tests if camera was released properly:
33. E.g., record a video, stop, then start another video right away. The camera should open again without error “camera already in use”.
34. Do the same for raw (take a raw, then another raw).
35. Especially test back-to-back video+raw sessions (to simulate continuous usage). Monitor memory (if possible via Android Studio profiler) to see that each session doesn’t leak (no growing memory or file handles).
36. **Error Handling Test:** Intentionally try some incorrect usage to see if it’s handled:
37. For example, disable RAW on a device that doesn’t support RAW and see if the module either automatically falls back or gives a controlled error (for S21/S22 this isn’t an issue, but if you had a lower device).
38. Turn off the screen or background the app while recording to check if your implementation needs to handle pause (depending on app requirements, camera might need to stop on pause). On Samsung devices, going home might not immediately stop camera, but if the app loses focus we might want to pause recording.
39. If possible, simulate an error: e.g., fill up device storage and then try recording to see how it fails (MediaRecorder might call onError or throw on stop). This is advanced, but good to know the app doesn’t crash and sessionInfo can note an error.
40. **Device Compatibility Check:** Although primarily for S21/S22:
41. Test on both an S21 and S22 if available. They have similar capabilities, but different chipsets (Exynos vs Snapdragon variants) which might have subtle differences. Ensure both can do RAW+4K. The test results should be the same.
42. If available, test on a lower device (Camera2 FULL but not level3, e.g., a Pixel 3a or something) to see what happens when trying RAW+video. Possibly it might still work (Pixel 3a might drop to 1080p if 4K not supported or slower frame rate). If it fails, ensure the app doesn’t crash – it should handle session config failure gracefully (maybe by disabling RAW or video and informing user).
43. Also test on a device with LEVEL\_3 that isn't Samsung (like a Pixel 5) to ensure our code is not Samsung-specific. This is more of a future-proofing test.
44. **Future Extension Stubs:** Though not implemented, consider testing the structure for extension:
45. e.g., if adding a still JPEG capture function, does the current design allow easily adding an ImageReader for JPEG? (This is more of a code review test rather than manual).
46. Or if a “calibration trigger” needs to capture a sequence of images, can it be done by leveraging the existing raw capture logic in a loop? This ensures the design is indeed modular.
47. **Performance Observations:** While testing, note:
    * Latency from clicking “start” to preview actually showing and recording starting – should be a second or less. Prolonged delays might mean an issue in threading or too much setup on main thread.
    * The smoothness of preview and lack of significant lag when capturing RAW during video – minor stutter can be acceptable but long freezes are problematic.
    * Device temperature if recording for a longer period (though a 10s test might not show it, but a 1-2 minute video+raw test could be done). S21/S22 can heat up with camera use; ensure no thermal throttling message or crash.

**Verification Criteria:** Each test above should pass without crashes or unexpected errors. All files (MP4, DNG) should be accessible and correct. If any test fails (e.g., RAW image is corrupt or video doesn’t save), that indicates a bug to fix in the implementation (for example, maybe forgetting to close the image or stop recorder properly).

By following this test plan, we can be confident the CameraRecorder meets the milestone requirements in real-world use. Documentation of results from these tests can accompany the implementation guide to show compliance with Milestone 2.2.

## Design Considerations for Modularity and Future Extensions

*(Beyond the asked points, summarizing how the design can accommodate future needs):*

The CameraRecorder is structured to isolate different concerns (preview, video recording, raw capture, etc.), which makes it easier to extend. For example, adding **still JPEG capture** would involve adding another ImageReader (JPEG format) and perhaps a new method capturePhoto(). Our session configuration can already handle multiple outputs, so we could include a JPEG surface when needed. The threading model and capture logic would remain largely the same, just with another path for handling JPEG in a similar way to RAW (but possibly using ImageWriter if doing reprocessing, or a simple capture for a photo).

For **calibration triggers** (perhaps capturing a pair of images or a special sequence for calibration), we can reuse raw capture or even add new routines: maybe a calibration requires a burst of RAW images – we could loop captureRawImage() calls or use captureBurst with a series of requests. The current design using Camera2 still supports session.captureBurst(List<CaptureRequest>, ...) which we could leverage in future.

For **focus control**: Since Camera2 allows manual focus or focus distance control, we can add public APIs to set focus modes or specific distances. The CameraRecorder could expose a method like setFocusMode(int mode) or triggerFocus(). Internally, this would use the captureSession by updating the repeating request builder – thanks to having stored the CaptureRequest.Builder for preview, we can adjust it (e.g., set AF\_MODE\_OFF and LENS\_FOCUS\_DISTANCE for manual focus, or call an AF trigger). The modular design with a persistent builder for preview (like mPreviewBuilder) means we can update and call session.setRepeatingRequest again to apply changes.

Finally, because we chose a flexible approach to sessions (closing and reopening as needed), it’s also possible to extend to use cases like **switching cameras** (if needed, to another lens), by stopping the session and reinitializing with a different camera ID. None of the logic is hard-coded to a specific camera beyond selection.

The key point is that each additional feature will mainly involve adding new surfaces or capture requests, which our structured approach can accommodate without a fundamental rewrite. This fulfills the goal of a clean, modular design for the synchronized recording platform beyond Milestone 2.2.

[[1]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=for%20%28String%20cameraId%20%3A%20manager,continue%3B) [[4]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=Size%20largestJpeg%20%3D%20Collections,closed%20when%20all%20background%20tasks) [[5]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=new%20CompareSizesByArea,RAW_SENSOR%29%29%2C%20new%20CompareSizesByArea) [[21]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=synchronized%20%28mCameraStateLock%29%20,5%29%29%3B) [[23]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=public%20void%20onOrientationChanged%28int%20orientation%29%20,) [[25]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=%2F%2F%20the%20SurfaceTextureListener%29,canDetectOrientation) [[26]](https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java#:~:text=configureTransform%28mTextureView.getWidth%28%29%2C%20mTextureView.getHeight%28%29%29%3B%20%7D%20else%20,mOrientationListener.enable%28%29%3B%20%7D) samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java - platform/development - Git at Google

<https://android.googlesource.com/platform/development/+/9ad662b8a0d0276cb437fed6a4121c27f9665a5a/samples/browseable/Camera2Raw/src/com.example.android.camera2raw/Camera2RawFragment.java>

[[2]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=,mTextureView.setAspectRatio%28mPreviewSize.getWidth%28%29%2C%20mPreviewSize.getHeight) [[3]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=StreamConfigurationMap%20map%20%3D%20characteristics%20,class%29%2C%20width%2C%20height%2C%20mVideoSize) [[7]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mCameraDevice.createCaptureRequest%28CameraDevice.TEMPLATE_RECORD%29%3B%20List,addTarget%28recorderSurface) [[8]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=,the%20camera%20preview%20size%20is) [[9]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=setUpCaptureRequestBuilder,thread.start%28%29%3B%20mPreviewSession.setRepeatingRequest%28mPreviewBuilder.build%28%29%2C%20null%2C%20mBackgroundHandler) [[10]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=%2F%2F%20UI%20mIsRecordingVideo%20%3D%20false%3B,%2B%20getVideoFile%28activity) [[11]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mMediaRecorder,startPreview) [[12]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mMediaRecorder,AAC) [[13]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mMediaRecorder,AAC) [[14]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mMediaRecorder,get%28rotation) [[15]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mMediaRecorder,get%28rotation) [[16]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=setUpMediaRecorder%28%29%3B%20SurfaceTexture%20texture%20%3D%20mTextureView,getHeight) [[17]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mIsRecordingVideo%20%3D%20true%3B%20%2F%2F%20Start,) [[22]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=texture) [[24]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=private%20void%20configureTransform,getHeight) [[27]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=int%20rotation%20%3D%20activity,bufferRect.centerY) [[28]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=if%20%28Surface.ROTATION_90%20%3D%3D%20rotation%20,2%29%2C%20centerX%2C%20centerY) [[29]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=mPreviewSize%20%3D%20chooseOptimalSize%28map,mTextureView.setAspectRatio%28mPreviewSize.getWidth%28%29%2C%20mPreviewSize.getHeight) [[30]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=if%20%28%21mCameraOpenCloseLock.tryAcquire%282500%2C%20TimeUnit.MILLISECONDS%29%29%20,SCALER_STREAM_CONFIGURATION_MAP) [[31]](https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java#:~:text=try%20,release%28%29%3B%20mMediaRecorder%20%3D%20null) samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java - platform/development - Git at Google

<https://android.googlesource.com/platform/development/+/abededd/samples/browseable/Camera2Video/src/com.example.android.camera2video/Camera2VideoFragment.java>

[[6]](https://developer.android.com/media/camera/camera2/multiple-camera-streams-simultaneously#:~:text=%2F%2F%20You%20will%20use%20the,TEMPLATE_PREVIEW) Use multiple camera streams simultaneously  |  Android media  |  Android Developers

<https://developer.android.com/media/camera/camera2/multiple-camera-streams-simultaneously>

[[18]](https://stackoverflow.com/questions/57126430/taking-a-dng-picture-using-the-camera2-api#:~:text=if%20%28mImage.format%20%3D%3D%20ImageFormat.RAW_SENSOR%29%20,finally) [[19]](https://stackoverflow.com/questions/57126430/taking-a-dng-picture-using-the-camera2-api#:~:text=,be%20saved%2C%20for%20example) [[20]](https://stackoverflow.com/questions/57126430/taking-a-dng-picture-using-the-camera2-api#:~:text=try%20,mImage.close%28%29%20closeOutput%28output%29) android - Taking a dng picture using the Camera2 API - Stack Overflow

<https://stackoverflow.com/questions/57126430/taking-a-dng-picture-using-the-camera2-api>