Chapter 1

Topdon TC001/TC001 Plus Android SDK — README

1.1 Overview

Topdon TC001 and TC001 Plus are portable infrared thermal cameras that attach to Android devices (via USB Type-C) to transform a smartphone or tablet into a high-tech thermal imager [?]. They capture heat radiation as images, allowing non-contact temperature measurement across a scene. The TC001 series features a 256×192 IR sensor resolution, producing clear thermal images and detecting temperature differences as small as 0.1°C [?] [?]. The TC001 Plus model includes a dual-lens system (an IR sensor plus a visible-light camera) enabling image fusion for sharper detail and contours in the thermal image [?] [?]. Both cameras support a wide temperature detection range from about **-20°C to 550°C** (-4°F to 1022°F) [?] [?], making them suitable for various applications — from industrial inspections to biomedical and physiological sensing.

In **physiological sensing**, infrared thermography provides a noninvasive, contact-free way to monitor subtle temperature changes on the body [?] [?]. Many physiological signals manifest as thermal patterns: for example, breathing rate can be measured by the cyclical temperature changes near the nostrils, heart pulse by tiny thermal pulsations, and stress-induced blood flow changes by temperature shifts in facial regions [?]. Indeed, thermal imaging has been used in psychophysiology to observe stress responses — e.g. a drop in nose tip temperature under acute stress or increased forehead temperature from vasodilation [?]. With the Topdon TC001 cameras, researchers can capture such thermal phenomena in real time, alongside other biosignals, for multimodal analysis of human physiological states.

1.2 Project Scope

This SDK/API provides an interface for Android developers to integrate the Topdon TC001/TC001 Plus camera into their applications. Its primary purpose is to facilitate **image acquisition, device configuration, and data streaming** from the thermal camera on Android. Using the SDK, an app can connect to the camera (via Android's USB host interface), start the ther-

mal video feed, and retrieve **thermal frames** (infrared images and temperature data) in real time. The API exposes methods to configure camera parameters — for example, selecting color palettes (pseudo-color schemes for the thermal image), adjusting image orientation, switching between high- and low-gain modes, triggering calibrations (shutter correction), and choosing the **data output mode** (e.g. image only, temperature only, or both interleaved). Under the hood, the SDK handles the low-level USB Video Class (UVC) protocol and infrared sensor commands, so developers can work with high-level objects (like a `UVCCamera` and frame callback) rather than raw USB transfers.

By using this SDK, developers can **stream thermal imagery** into their apps for visualization or analysis, obtain per-pixel temperature readings, and synchronously capture frames at up to 25 Hz (depending on device) [**?**]. The SDK is designed to support **both** the TC001 and TC001 Plus models, abstracting their dual-lens or single-lens differences. For instance, when a TC001 Plus is connected, the SDK can fetch the simultaneous visual and IR streams for fusion, whereas for TC001 it handles the single thermal stream. Overall, the project enables Android applications to utilize the TC001 series cameras for tasks such as real-time thermal monitoring, recording thermal videos, measuring temperature at points or regions of interest, and integrating thermal data with other sensor modalities.

1.3 Installation

To integrate the Topdon TC001 SDK into an Android project, begin by obtaining the SDK package. Topdon provides an Android SDK (e.g. as a `.zip` archive) containing the library binaries and sample code [**?**]. This typically includes a precompiled AAR library (or JAR with native `.so` files) and documentation. Import the SDK library into your Android Studio project by copying the AAR into your app module's `libs` folder and adding it as a dependency in your Gradle build file. For example, in `app/build.gradle`:

dependencies implementation fileTree(dir: "libs", include: [".*aar"]) // ... other dependencies ...

Make sure to enable support for the USB host API in your app's manifest (the Topdon camera uses Android's USB host mode). In the app `AndroidManifest.xml`, declare the USB host feature:

<uses-feature android:name="android.hardware.usb.host" android:required="true"/>

Also add an intent filter and metadata so that Android can recognize the camera and grant permissions. For example, inside your launch Activity:

<intent-filter> <action android:name="android.hardware.usb.action.USB$_{D}EVICE_{A}TTACHED$"/><
$/intent-filter><meta-dataandroid:name=$"$android.hardware.usb.action.USB_{D}EVICE_{A}TTACHED$
$resource=$"@$xml/device_{f}ilter$"/>

Here, `@xml/device`$_{f}$`ilter`$is a resource file (placed in `res/xml/`) specifying the USB Vendor ID and Product ID$

`Using the TC001 camera on Android requires a few permissions and configurations:`

- `USB Permissions: The camera communicates via USB, so your app must request permission to access the USB device. The Android system will prompt the user with a dialog`

such as "Allow this app to access the USB device?". The SDK (via USBMonitor) handles this by calling UsbManager.requestPermission() when the device is attached. Ensure your manifest includes the USB device intent filter as shown above and that your app logic requests/grants permission. (There is no

- Camera Permission: Even though the TC001 is an external camera, the system may treat it as a camera source. It is advisable to declare the camera permission in your manifest and request it on devices running Android 6.0+ for completeness. For example: <uses-permission android:name="android.permission.CAMERA" /> [?].

- Storage Permissions (optional): If your application will save thermal images or videos to device storage, include read/write storage permissions. The SDK sample app requests:

- WRITE$_EXTERNAL_STORAGEand$ In summary, add the following to your AndroidManifest.xml under the <manifest> node:

  <uses-permission android:name="android.permission.CAMERA"/> <uses-permission android:name=
  $uses-permissionandroid:name = "android.permission.READ_EXTERNAL_STORAGE"/>$

  And ensure the USB host feature is declared as mentioned. The user will need to confirm the USB permission each time the camera is connected (unless your app is pre-installe as a system app), so be prepared to handle the permission grant flow.

1.4 Getting Started

Once the SDK is integrated and permissions are in place, you can initialize and connect to the TC001 camera in your app. The basic workflow is:

1. Initialize the USB monitor and listeners. The SDK provides a USBMonitor class that monitors USB device connections. You attach an OnDeviceConnectListener to handle events: device attached, device permission granted, device connected, disconnected, etc.

2. Request permission and open the camera. When the camera is attached and permission is granted, create a UVCCamera instance using the SDK's builder, open it with the USB control block, and initialize the infrared command interface (IRCMD) for thermal data.

3. Start streaming frames. Set a frame callback to receive image frames, then start the camera preview. As frames come in, you can process the thermal image and/or temperature data.

4. Close the camera on disconnect. Properly stop streaming and release resources if the camera is detached.

Below is a sample code snippet illustrating the initialization and frame capture process in Java:

import android.hardware.usb.UsbDevice; import com.infisense.iruvc.usb.USBMonitor; import com.infisense.iruvc.uvc.UVCCamera; import com.infisense.iruvc.uvc.UVCType; import com.infisense.iruvc.uvc.ConnectCallback; import com.infisense.iruvc.ircmd.IRCMD;

import com.infisense.iruvc.ircmd.IRCMDType; import com.infisense.iruvc.utils.CommonParams; import com.infisense.iruvc.utils.IFrameCallback; import com.infisense.iruvc.uvc.ConcreateUVCB import com.infisense.iruvc.ircmd.ConcreteIRCMDBuilder;

// ... inside an Activity or Service:

USBMonitor usbMonitor = new USBMonitor(getApplicationContext(), new USBMonitor.OnDeviceCon @Override public void onAttach(UsbDevice device) // Called when a USB device (camera) is attached usbMonitor.requestPermission(device); // request user permission @Override public void onGranted(UsbDevice device, boolean granted) // Called after user grants/denies permission (not used here, we wait for onConnect) @Override public void onConnect(UsbDevice device, USBMonitor.UsbControlBlock ctrlBlock, boolean createNew) if (!createNew) return; // Permission granted and new device connection: // 1. Initialize the UVC camera object ConcreateUVCBuilder uvcBuilder = new ConcreateUVCBuilder(); UVCCamera uvcCamera = uvcBuilder .setUVCType(UVCType.USB$_UVC$)//$usingaUSBUVCdevice.build(); intresult = uvcCamera.openUVCCamera(ctrlBlock); //openthecameraif(result! = 0)//handleerror(resultcodesdefine newConcreteIRCMDBuilder().setIrcmdType(IRCMDType.USB_IR_2$56_3$84)//using256x384IRmodule(for

// Register the USBMonitor to start listening for events usbMonitor.register(); // (Don't forget to unregister in onPause/onDestroy to avoid leaks)

In the above code:

- We create a USBMonitor with an OnDeviceConnectListener. In onAttach, we immediately request permission for the device. When permission is granted, onConnect is invoked with a UsbControlBlock that we use to open the camera.
- A UVCCamera is built via ConcreateUVCBuilder (the SDK's builder for UVC-compliant cameras) and opened. We then build an IRCMD object via ConcreteIRCMDBuilder -- this represents the infrared command interface, which handles thermal sensor configuration and commands. We specify IRCMDType.USB$_I R_2 56_3 84 assuminga 256192 sensorwithim temperature(256384 outputframe)1001[?]1001[?].(TheSDKmayhavedifferentIRCMDTypeenumsifother cameramodesaresupported.)WesetanIFrameCallbackonthecamera.TheSDKwillinvokeonFrame(byte[

- Finally, we call uvcCamera.onStartPreview() and then use irCmd.startPreview(...) to begin the infrared data stream [?] [?]. We pass parameters specifying the source (sensor), target frame rate, mode (here using VOC$_D VP_M ODE, adefaultmodeforinfraredoutput), an$ Note: All USBMonitor callbacks (onAttach, onConnect, etc.) run on a background thread [?]. This means the onFrame callback is also on a background thread, so you should handle thread synchronization if updating UI elements with the thermal image. In a real app, you might use a handler or runOnUiThread to display the image, or process frames in a dedicated worker thread (as the SDK sample does with an ImageThread for image conversion).

This basic initialization can be adapted. For example, if you only need the temperature data without the image, you could use DataFlowMode.TEMP$_O UTPUT whenstartingpreview.Orifyouonl$

Frame Format: The TC001 outputs thermal frames in a YUV format (specifically YUYV 4:2:2) by default for the infrared image [?]. When using IMAGE$_A ND_T EMP_O UTPUT mode(asinth$

$-The first half of the byte array is the infrared image data (thermal image) in YUV422 format. -The second half of$
$pixel temperature data, also in a 16-bit format (Y16).$

For the TC001's 256×192 resolution: if both image and temperature are enabled, the frame is effectively 256×384 in dimension (the two 256×192 images stacked). In memory this translates to a byte array length of 256 384 *2 = 196608* bytes, since YUV422 uses 2 bytes per pixel on average. The first 98,304 bytes correspond to a 256×192 YUV image, and the next 98,304 bytes correspond to a 256×192 array of temperature values [?] [?]. The SDK documentation confirms that "the frame array's front half is IR data, the latter half is temperature data" for image+temperature mode [?] [?]. If you use image-only or temp-only modes, the frame will be half that size, containing just the single dataset.

Converting the Image: To display the thermal image, you need to convert the YUYV data to a viewable format (e.g., ARGB8888). The SDK includes a LibIRProcess utility with methods like convertYuyvMapToARGBPseudocolor(...) and convertArrayYuv422ToARGB(...) [?]. In the provided sample, the raw YUV is converted to an ARGB bitmap as follows (pseudo-code):

// Assume imageBytes is the first half of frameData containing YUV422 data int pixelCount = imageWidth * imageHeight; // 256*192 int[] argbPixels = new int[pixelCount]; if (pseudocolorMode != null) // Apply a false-color palette to the grayscale thermal image LibIRProcess.convertYuyvMapToARGBPseudocolor(imageBytes, (long)pixelCount, pseudocolorM argbPixels); else // Convert YUV422 to grayscale ARGB LibIRProcess.convertYuv422ToARGB(imag pixelCount, argbPixels); // If the image is rotated (camera orientation), rotate accordingly: if (needRotate90) LibIRProcess.ImageRes$_t res = new LibIRProcess.ImageRes_t(); res.h$
$(char) imageWidth; res.width = (char) imageHeight; //Rotate right by 90 degrees LibIRProcess.rotateRight90$
$Bitmap thermalBitmap = Bitmap.createBitmap(argbPixels, imageWidth, imageHeight, Bitmap.Config.A$

This process is essentially what the SDK's sample ImageThread does: YUV - ARGB - rotate - Bitmap [?] [?]. You can then draw this Bitmap on an ImageView or a custom view (CameraView in the sample) to show the live thermal image. The SDK supports multiple pseudo-color palettes (ironbow, rainbow, grayscale, etc.), approximately 10 palettes including "white hot", "black hot", "medical" and others [?]. By selecting a palette (e.g., CommonParams.PseudoColorType), you can have the conversion routine apply that coloring to the image, enhancing contrast for visualization.

Temperature Data: The temperature values for each pixel are provided in the second half of the frame (if using combined mode, or as the whole frame in temp-only mode). These are 16-bit raw data values that correspond to temperatures in either Kelvin or Celsius with a scaling factor. According to the SDK, in Y16 format each unit corresponds to 1/64 of a degree (for 16-bit data) [?]. In other words, to convert a raw 16-bit value to an absolute temperature:

- If using the default scale (for TC001 high-gain mode), divide the raw value by 64 to get temperature in Kelvin, then subtract 273.15 to convert to °C. For example, a raw value of 30000 would be 30000/64 = 468.75 K, which is 195.6 °C.
- If the camera or mode uses 14-bit data (scale 16, e.g. low-gain mode or IR fusion

mode), divide by 16 instead [?]. The SDK's LibIRTemp class handles these conversions internally by setting the scale depending on mode (64 for Y16, 16 for Y14) [?].

Typically, you won't manually convert every pixel unless needed. A common approach is to use provided methods to find min/max or get specific points. The LibIRTemp class can be used to extract temperature metrics from the byte array. For instance, you can instantiate LibIRTemp with the image width/height and then call methods to sample temperatures:

LibIRTemp irTempUtil = new LibIRTemp(imageWidth, imageHeight); irTempUtil.setTempData(temp LibIRTemp.TemperatureSampleResult result = irTempUtil.getTemperatureOfRect(new Rect(0,0, imageWidth-1, imageHeight-1)); float minC = result.minTemperature; // minimum temperature in °C in the image Point minLoc = result.minTemperaturePixel; // location of min temp float maxC = result.maxTemperature;

The SDK can compute statistics like max, min, and average temperature over regions (rectangles, lines, or points) [?] [?]. The sample's TemperatureView uses these methods to display values for user-selected points/areas on the image (up to 3 points, lines, or rectangles) -- for example, it draws the hottest and coldest point in a region with their temperature values [?] [?]. Using these utilities ensures the proper calibration and scale is applied (the SDK accounts for the camera's calibration data and any offset). The accuracy of the temperature readings is stated as ±2°C or ±2series [?] [?], and the noise-equivalent temperature difference (NETD) is 40 mK, meaning very small temperature differences (Õ.04 °C) are distinguishable [?].

When visualizing the temperature data, you might overlay it as numeric values or a separate "temperature map". The TC001 Plus, with its visual camera, allows blending the thermal data with an RGB image -- the SDK can provide an aligned visual image so you can overlay temperature info on actual visible contours [?]. In our context, if focusing on physiological data, one might not need the visible-light overlay, but it can help identify anatomical features (the SDK sample synchronizes RGB and thermal frames to locate facial landmarks in thermal images [?] [?]).

Thermal Range and Units: As mentioned, the device can measure roughly -20 °C to 550 °C in two gain modes [?]. The camera will automatically switch between high gain (for lower temperatures, finer resolution) and low gain (for higher temperatures, extended range). The SDK provides callbacks (AutoGainSwitch) if one enables them, but by default this is handled internally. The raw values and scaling differ slightly in low-gain mode (14-bit data); the LibIRTemp.setScale(16) call in the sample is doing exactly that when IRISP (the dual-gain fusion mode) is on [?]. For most physiological use-cases (human body temperatures Õ30--40 °C), the camera will be in high-gain mode for better sensitivity.

In summary, handling data from the SDK involves splitting the frame into image vs. temperature, converting the image for display (using provided conversion functions and applying pseudo-colors if desired), and converting temperature data to meaningful values (using the scale or the LibIRTemp helpers). The provided sample code and SDK documentation have detailed examples of these steps, ensuring you can both see the

thermal scene and measure temperatures at points of interest.

1.5 Integration with bucika$_gsr$

The bucika$_gsr$ $Android app is a multi-modal data collection tool, combining galvanic skin response (GSR) with oth$

Architecture Fit: In a typical setup, the app might have a background service or manager handling sensor data acquisition. The TC001 integration would add a Thermal Sensor Module to this system. For example, when a recording session starts, the app would:

- Power and connect to the TC001 camera (using the SDK as outlined in *specific <uses-permission> string for USB host; it's enabled by the uses-feature tag and user consent dialog.)*
- *Camera Permission: Even though the TC001 is an external camera, the system may treat it as a camera source. It is advisable to declare the camera permission in your manifest and request it on devices running Android 6.0+ for completeness. For example: <uses-permission android:name="android.permission.CAMERA" /> [?].*
- *Storage Permissions (optional): If your application will save thermal images or videos to device storage, include read/write storage permissions. The SDK sample app requests:*

- $WRITE_EXTERNAL_STORAGE and$ In summary, add the following to your AndroidManifest.xml under the <manifest> node:

<uses-permission android:name="android.permission.CAMERA"/> <uses-permission android:name=

$uses-permission android : name = "android.permission.READ_EXTERNAL_STORAGE" / >$

And ensure the USB host feature is declared as mentioned. The user will need to confirm the USB permission each time the camera is connected (unless your app is pre-installe as a system app), so be prepared to handle the permission grant flow.

1.6 Getting Started

Once the SDK is integrated and permissions are in place, you can initialize and connect to the TC001 camera in your app. The basic workflow is:

1. Initialize the USB monitor and listeners. The SDK provides a USBMonitor class that monitors USB device connections. You attach an OnDeviceConnectListener to handle events: device attached, device permission granted, device connected, disconnected, etc.

2. Request permission and open the camera. When the camera is attached and permission is granted, create a UVCCamera instance using the SDK's builder, open it with the USB control block, and initialize the infrared command interface (IRCMD) for thermal data.

3. Start streaming frames. Set a frame callback to receive image frames, then start the camera preview. As frames come in, you can process the thermal image and/or temperature data.

4. Close the camera on disconnect. Properly stop streaming and release resources if the camera is detached.

Below is a sample code snippet illustrating the initialization and frame capture process in Java:

import android.hardware.usb.UsbDevice; import com.infisense.iruvc.usb.USBMonitor; import com.infisense.iruvc.uvc.UVCCamera; import com.infisense.iruvc.uvc.UVCType; import com.infisense.iruvc.uvc.ConnectCallback; import com.infisense.iruvc.ircmd.IRCMD; import com.infisense.iruvc.ircmd.IRCMDType; import com.infisense.iruvc.utils.CommonParams; import com.infisense.iruvc.utils.IFrameCallback; import com.infisense.iruvc.uvc.ConcreateUVCB import com.infisense.iruvc.ircmd.ConcreteIRCMDBuilder;

// ... inside an Activity or Service:

USBMonitor usbMonitor = new USBMonitor(getApplicationContext(), new USBMonitor.OnDeviceCon @Override public void onAttach(UsbDevice device) // Called when a USB device (camera) is attached usbMonitor.requestPermission(device); // request user permission @Override public void onGranted(UsbDevice device, boolean granted) // Called after user grants/denies permission (not used here, we wait for onConnect) @Override public void onConnect(UsbDevice device, USBMonitor.UsbControlBlock ctrlBlock, boolean createNew) if (!createNew) return; // Permission granted and new device connection: // 1. Initialize the UVC camera object ConcreateUVCBuilder uvcBuilder = new ConcreateUVCBuilder(); UVCCamera uvcCamera = uvcBuilder .setUVCType(UVCType.USB$_UVC$)//$using a USB UVC device.build(); int result = uvcCamera.open UVC Camera(ctrlBlock); //open the camera if (result != 0)//handle error (result codes define new Concrete IRCMD Builder().set Ircmd Type(IRCMDType.USB$_I R_2 56_3 84$)//$using 256x384 IR module (for$

// Register the USBMonitor to start listening for events usbMonitor.register(); // (Don't forget to unregister in onPause/onDestroy to avoid leaks)

In the above code:

- We create a USBMonitor with an OnDeviceConnectListener. In onAttach, we immediately request permission for the device. When permission is granted, onConnect is invoked with a UsbControlBlock that we use to open the camera.
- A UVCCamera is built via ConcreateUVCBuilder (the SDK's builder for UVC-compliant cameras) and opened. We then build an IRCMD object via ConcreteIRCMDBuilder -- this represents the infrared command interface, which handles thermal sensor configuration and commands. We specify IRCMDType.USB$_I R_2 56_3 84 assuming a 256192 sensor with im temperature$(256384 output frame)1001[?]1001[?].$(The SDK may have different$IRCMDType$enums if other camera modes are supported.) We set an$IFrameCallback$on the camera. The SDK will invoke$onFrame(byte[

- Finally, we call uvcCamera.onStartPreview() and then use irCmd.startPreview(...) to begin the infrared data stream [?] [?]. We pass parameters specifying the source (sensor), target frame rate, mode (here using VOC$_D VP_M ODE$, a default mode for infrared output), an Note: All USBMonitor callbacks (onAttach, onConnect, etc.) run on a background thread

[?]. This means the onFrame callback is also on a background thread, so you should handle thread synchronization if updating UI elements with the thermal image. In a real app, you might use a handler or runOnUiThread to display the image, or process frames in a dedicated worker thread (as the SDK sample does with an ImageThread for image conversion).

This basic initialization can be adapted. For example, if you only need the temperature data without the image, you could use DataFlowMode.TEMP$_OUTPUTwhenstartingpreview.Orifyouonl$

Frame Format: The TC001 outputs thermal frames in a YUV format (specifically YUYV 4:2:2) by default for the infrared image [?]. When using IMAGE$_AND_TEMP_OUTPUTmode(asinth$ $-Thefirsthalfofthebytearrayistheinfraredimagedata(thermalimage)inYUV422format.-Thesecondhalfo$ $pixeltemperaturedata,alsoina16-bitformat(Y16).$

For the TC001's 256×192 resolution: if both image and temperature are enabled, the frame is effectively 256×384 in dimension (the two 256×192 images stacked). In memory this translates to a byte array length of 256 384 *2 = 196608* bytes, since YUV422 uses 2 bytes per pixel on average. The first 98,304 bytes correspond to a 256×192 YUV image, and the next 98,304 bytes correspond to a 256×192 array of temperature values [?] [?]. The SDK documentation confirms that "the frame array's front half is IR data, the latter half is temperature data" for image+temperature mode [?] [?]. If you use image-only or temp-only modes, the frame will be half that size, containing just the single dataset.

Converting the Image: To display the thermal image, you need to convert the YUYV data to a viewable format (e.g., ARGB8888). The SDK includes a LibIRProcess utility with methods like convertYuyvMapToARGBPseudocolor(...) and convertArrayYuv422ToARGB(...) [?]. In the provided sample, the raw YUV is converted to an ARGB bitmap as follows (pseudo-code):

// Assume imageBytes is the first half of frameData containing YUV422 data int pixelCount = imageWidth * imageHeight; // 256*192 int[] argbPixels = new int[pixelCount]; if (pseudocolorMode != null) // Apply a false-color palette to the grayscale thermal image LibIRProcess.convertYuyvMapToARGBPseudocolor(imageBytes, (long)pixelCount, pseudocolorM argbPixels); else // Convert YUV422 to grayscale ARGB LibIRProcess.convertYuv422ToARGB(imag pixelCount, argbPixels); // If the image is rotated (camera orientation), rotate accordingly: if (needRotate90) LibIRProcess.ImageRes$_t res = newLibIRProcess.ImageRes_t(); res.he$ $(char)imageWidth; res.width = (char)imageHeight; //Rotaterightby90degreesLibIRProcess.rotateRight90$ $BitmapthermalBitmap = Bitmap.createBitmap(argbPixels, imageWidth, imageHeight, Bitmap.Config.A$

This process is essentially what the SDK's sample ImageThread does: YUV - ARGB - rotate - Bitmap [?] [?]. You can then draw this Bitmap on an ImageView or a custom view (CameraView in the sample) to show the live thermal image. The SDK supports multiple pseudo-color palettes (ironbow, rainbow, grayscale, etc.), approximately 10 palettes including "white hot", "black hot", "medical" and others [?]. By selecting a palette (e.g., CommonParams.PseudoColorType), you can have the conversion routine apply that coloring to the image, enhancing contrast for visualization.

Temperature Data: The temperature values for each pixel are provided in the second

half of the frame (if using combined mode, or as the whole frame in temp-only mode). These are 16-bit raw data values that correspond to temperatures in either Kelvin or Celsius with a scaling factor. According to the SDK, in Y16 format each unit corresponds to 1/64 of a degree (for 16-bit data) [?]. In other words, to convert a raw 16-bit value to an absolute temperature:

- If using the default scale (for TC001 high-gain mode), divide the raw value by 64 to get temperature in Kelvin, then subtract 273.15 to convert to °C. For example, a raw value of 30000 would be 30000/64 = 468.75 K, which is 195.6 °C.
- If the camera or mode uses 14-bit data (scale 16, e.g. low-gain mode or IR fusion mode), divide by 16 instead [?]. The SDK's LibIRTemp class handles these conversions internally by setting the scale depending on mode (64 for Y16, 16 for Y14) [?].

Typically, you won't manually convert every pixel unless needed. A common approach is to use provided methods to find min/max or get specific points. The LibIRTemp class can be used to extract temperature metrics from the byte array. For instance, you can instantiate LibIRTemp with the image width/height and then call methods to sample temperatures:

LibIRTemp irTempUtil = new LibIRTemp(imageWidth, imageHeight); irTempUtil.setTempData(temp LibIRTemp.TemperatureSampleResult result = irTempUtil.getTemperatureOfRect(new Rect(0,0, imageWidth-1, imageHeight-1)); float minC = result.minTemperature; // minimum temperature in °C in the image Point minLoc = result.minTemperaturePixel; // location of min temp float maxC = result.maxTemperature;

The SDK can compute statistics like max, min, and average temperature over regions (rectangles, lines, or points) [?] [?]. The sample's TemperatureView uses these methods to display values for user-selected points/areas on the image (up to 3 points, lines, or rectangles) -- for example, it draws the hottest and coldest point in a region with their temperature values [?] [?]. Using these utilities ensures the proper calibration and scale is applied (the SDK accounts for the camera's calibration data and any offset). The accuracy of the temperature readings is stated as ±2°C or ±2series [?] [?], and the noise-equivalent temperature difference (NETD) is 40 mK, meaning very small temperature differences (Õ.04 °C) are distinguishable [?].

When visualizing the temperature data, you might overlay it as numeric values or a separate "temperature map". The TC001 Plus, with its visual camera, allows blending the thermal data with an RGB image -- the SDK can provide an aligned visual image so you can overlay temperature info on actual visible contours [?]. In our context, if focusing on physiological data, one might not need the visible-light overlay, but it can help identify anatomical features (the SDK sample synchronizes RGB and thermal frames to locate facial landmarks in thermal images [?] [?]).

Thermal Range and Units: As mentioned, the device can measure roughly -20 °C to 550 °C in two gain modes [?]. The camera will automatically switch between high gain (for lower temperatures, finer resolution) and low gain (for higher temperatures, extended range). The SDK provides callbacks (AutoGainSwitch) if one enables them,

but by default this is handled internally. The raw values and scaling differ slightly in low-gain mode (14-bit data); the LibIRTemp.setScale(16) call in the sample is doing exactly that when IRISP (the dual-gain fusion mode) is on [?]. For most physiological use-cases (human body temperatures $\tilde{3}$0--40 °C), the camera will be in high-gain mode for better sensitivity.

In summary, handling data from the SDK involves splitting the frame into image vs. temperature, converting the image for display (using provided conversion functions and applying pseudo-colors if desired), and converting temperature data to meaningful values (using the scale or the LibIRTemp helpers). The provided sample code and SDK documentation have detailed examples of these steps, ensuring you can both see the thermal scene and measure temperatures at points of interest.

1.7 Integration with bucika$_gsr$

The bucika$_gsr$ $Android app is a multi-modal data collection tool, combining galvanic skin response (GSR) with oth$

Architecture Fit: In a typical setup, the app might have a background service or manager handling sensor data acquisition. The TC001 integration would add a Thermal Sensor Module to this system. For example, when a recording session starts, the app would:

- Power and connect to the TC001 camera (using the SDK as outlined in Getting Started
- Begin streaming thermal frames. You may choose to store the raw thermal video or to compute specific metrics in real-time. In a GSR-focused experiment, one might extract features like *). This could be done in the same service that reads GSR, or in a dedicated thread, as long as lifecycle is managed (start/stop with the session).*
- *Begin streaming thermal frames. You may choose to store the raw thermal video or to compute specific metrics in real-time. In a GSR-focused experiment, one might extract features like facial temperature averages, nose tip temperature over time, or rate of change of temperature*
- *Ensure that each thermal sample or frame is timestamped (using the same clock or reference as the GSR data timestamps). Given GSR is often sampled at, say, 10--100 Hz and the thermal camera at up to 25 Hz, you might down-sample or interpolate as needed. The sample research system described by Gioia to indicate stress responses [?]. These features can be computed on each frame or every few frames and then logged alongside GSR data.*
- *Ensure that each thermal sample or frame is timestamped (using the same clock or reference as the GSR data timestamps). Given GSR is often sampled at, say, 10--100 Hz and the thermal camera at up to 25 Hz, you might down-sample or interpolate as needed. The sample research system described by Gioia et al.*

- *Data Fusion: The GSR signal primarily reflects sympathetic nervous system arousal (sweat gland activity), whereas the thermal camera can capture peripheral blood flow changes and other heat-related signals [?] [?]. In bucika$_g$sr, thethermalstreamcouldbe relatedvasoconstriction)*1001[?].*Byintegratingthetwo, theappcoulddetectsuchpatternsmorereliably.The*

- *Real-Time Display: If bucika$_g$srhasauserinterfaceshowinglivesignals(suchasagraphofGSRovert time.TheTC*001*Plus′sduallenscanassistinaimingthecameraatthesubject′sfaceorskinareausingthevisi*

From a software integration standpoint, adding the TC001 likely means managing the lifecycle within the app: initializing the camera when needed, handling user permissions (the app might prompt the user to connect/allow the camera at start of a session), and gracefully stopping the camera when the session ends or app pauses. This involves calling usbMonitor.register() on start and usbMonitor.unregister() on stop, and releasing the camera (uvcCamera.close() etc.) to free the USB interface. The bucika$_g$*srappshouldalsoaccountforcaseswherethecameraisnotavailableortheuserdeniespermission, byf* In terms of multimodal data architecture, the thermal stream will produce a high volume of data (frame bytes or extracted features), so consider the data handling pipeline: you might not want to log every pixel's temperature for long sessions due to storage and processing load. Instead, the integration can focus on key features relevant to the research questions. Common choices in research are: maximum facial temperature, temperature at the tip of the nose, or at the inner canthus of the eyes (as indicators of stress/fear) [?], or overall average skin temperature as an indicator of thermal comfort. The SDK makes it easy to compute these in real-time (using LibIRTemp.getTemperature or small regions). Those values (a few numbers per second) can then be time-stamped and recorded alongside GSR and perhaps other signals (heart rate, etc.) in the app's data file.

By integrating the Topdon SDK with bucika$_g$*sr, theappeffectivelybecomesa*multisensor platform, *c inducedGSRfluctuations*1001[?]1001[?].*Suchintegrationopensthedoortomorerobustandcontactlessstressoren GSRrequiresskincontactelectrodes, whereasthermaliscontact−free−−−combiningthemcanvalidatefinding* 1001[?]1001[?].

In summary, within the bucika$_g$*srapp, theTopdoncamera′sroleistoprovidea*thermal imaging stream*th*

1.8 Troubleshooting

When working with the TC001/TC001 Plus on Android, you may encounter some common issues. Here are troubleshooting tips and solutions:

- Device Not Detected by the App: If plugging in the camera does nothing, ensure that your phone supports USB OTG (On-The-Go) and that the USB host feature is declared in the app manifest. Some phones require enabling OTG in settings or have power-saving modes that disable it after a period. Also verify your device*filter.xm Cconnectorisfullyinserted−−−thecamerashouldfirmlyattach, andonsomedevicesyoumightneedtoflip specific*).

- Permission Denied or No Prompt: If you never see the "Allow access" dialog, it could be because another app (or a system service) has claimed the USB interface. Make sure no other app (including the default Topdon app or other camera apps) is running and auto-opening the device. Also confirm your intent filter is correct. In some cases, you might need to manually call UsbManager.requestPermission(device, pendingIntent) if the USBMonitor approach is not triggering. The SDK's USBMonitor will broadcast an intent with action USBMonitor.ACTION$_{USB}PERMISSION$ — ensure your activity is registered to receive it (the SDK's sample takes care of this internally).

- Frame Freezes or No Image Output: If the camera connects but you only see a black screen or one static frame, it could be a bandwidth or mode issue. The SDK allows adjusting USB bandwidth: uvcCamera.setDefaultBandwidth(1.0f) is used in the sample to request maximum bandwidth [?]. Some Android devices have limited USB bandwidth for external cameras. Try setting a lower bandwidth factor (closer to 0) if frames aren't coming through, or reduce the frame rate parameter in startPreview() (e.g., try 9 or 10 fps to see if that stabilizes output). Also, ensure that the data flow mode you request is supported by the camera: TC001 should support the combined mode by default, but if you accidentally use a mode not supported (e.g., a higher resolution or a dual-camera mode on a single-camera device), it may not output. Use uvcCamera.getSupportedSizeList() to log what resolutions and modes the camera provides [?].

- Heat or Calibration Issues: The camera has an internal shutter that periodically calibrates the sensor (you might hear a soft *synchronized 5 Hz RGB frames with 25 Hz thermal frames for analysis [?], so a similar strategy can be applied: for instance, record thermal data at its native rate and later align it to GSR's timeline via timestamps.*

- *Data Fusion: The GSR signal primarily reflects sympathetic nervous system arousal (sweat gland activity), whereas the thermal camera can capture peripheral blood flow changes and other heat-related signals [?] [?]. In bucika$_{g}sr$, the thermal stream could be* $related vasoconstriction) 1001 [?]. By integrating the two, the app could detect such patterns more reliably. The$

- *Real-Time Display: If bucika$_{g}sr$ has a user interface showing live signals (such as a graph of GSR over t* $time. The TC001 Plus's dual lens can assist in aiming the camera at the subject's face or skin area using the visi$

From a software integration standpoint, adding the TC001 likely means managing the lifecycle within the app: initializing the camera when needed, handling user permissions (the app might prompt the user to connect/allow the camera at start of a session), and gracefully stopping the camera when the session ends or app pauses. This involves calling usbMonitor.register() on start and usbMonitor.unregister() on stop, and releasing the camera (uvcCamera.close() etc.) to free the USB interface. The bucika$_{g}sr$ app should also acc

In terms of multimodal data architecture, the thermal stream will produce a high volume of data (frame bytes or extracted features), so consider the data handling pipeline: you might not want to log every pixel's temperature for long sessions due

to storage and processing load. Instead, the integration can focus on key features relevant to the research questions. Common choices in research are: maximum facial temperature, temperature at the tip of the nose, or at the inner canthus of the eyes (as indicators of stress/fear) [?], or overall average skin temperature as an indicator of thermal comfort. The SDK makes it easy to compute these in real-time (using LibIRTemp.get or small regions). Those values (a few numbers per second) can then be time-stamped and recorded alongside GSR and perhaps other signals (heart rate, etc.) in the app's data file.

By integrating the Topdon SDK with bucika$_gsr, theappeffectivelybecomesa$multisensor platform, $cinducedGSRfluctuations$1001[?]1001[?]. $Suchintegrationopensthedoortomorerobustandcontactlessstressoren$ $GSRrequiresskincontactelectrodes, whereasthermaliscontact-free---combiningthemcanvalidatefinding$ 1001[?]1001[?].

In summary, within the bucika$_gsrapp, theTopdoncamera'sroleistoprovidea$thermal imaging stream$th$

1.9 Troubleshooting

When working with the TC001/TC001 Plus on Android, you may encounter some common issues. Here are troubleshooting tips and solutions:

- Device Not Detected by the App: If plugging in the camera does nothing, ensure that your phone supports USB OTG (On-The-Go) and that the USB host feature is declared in the app manifest. Some phones require enabling OTG in settings or have power-saving modes that disable it after a period. Also verify your device$_filter.xm$ $Cconnectorisfullyinserted---thecamerashouldfirmlyattach, andonsomedevicesyoumightneedtoflip$ $specific$).

- Permission Denied or No Prompt: If you never see the "Allow access" dialog, it could be because another app (or a system service) has claimed the USB interface. Make sure no other app (including the default Topdon app or other camera apps) is running and auto-opening the device. Also confirm your intent filter is correct. In some cases, you might need to manually call UsbManager.requestPermission(device, pendingIntent) if the USBMonitor approach is not triggering. The SDK's USBMonitor will broadcast an intent with action USBMonitor.ACTION$_USBPERMISSION---$ $ensureyouractivityisregisteredtoreceiveit(theSDK'ssampletakescareofthisinternally)$.

- Frame Freezes or No Image Output: If the camera connects but you only see a black screen or one static frame, it could be a bandwidth or mode issue. The SDK allows adjusting USB bandwidth: uvcCamera.setDefaultBandwidth(1.0f) is used in the sample to request maximum bandwidth [?]. Some Android devices have limited USB bandwidth for external cameras. Try setting a lower bandwidth factor (closer to 0) if frames aren't coming through, or reduce the frame rate parameter in startPreview() (e.g., try 9 or 10 fps to see if that stabilizes output). Also, ensure that the data flow mode you request is supported by the camera: TC001 should support the combined mode by default, but if you accidentally use a mode

not supported (e.g., a higher resolution or a dual-camera mode on a single-camera device), it may not output. Use uvcCamera.getSupportedSizeList() to log what resolutions and modes the camera provides [?].

- Heat or Calibration Issues: The camera has an internal shutter that periodically calibrates the sensor (you might hear a soft click *every so often). During those moments (which typically last a fraction of a second), the image may pause or show a uniform field. This is normal -- the camera is self-correcting for drift. If you find the calibration happening too often or at inconvenient times, you can manually trigger it at a known safe moment using ircmd.updateOOCOrB(CommonParams.Upd* $heating$

- *USB Connection Stability: If the camera frequently disconnects or frames stop, it could be a power issue -- the camera draws power from the phone. Ensure the phone is sufficiently charged and not in a low-power mode. Some users use a Y-cable to supply external power if needed, but for TC001 this is rarely required as power draw is modest (Õ.5 W). However, avoid having multiple high-power USB devices at once. Additionally, use the short OTG adapter that comes with the camera (or a high-quality cable) -- long or poor-quality cables can cause voltage drop or data errors. If you detect frame errors or the SDK flags a bad frame (the sample checks the last byte of the frame for a bad-frame flag [?]), it will attempt a sensor restart. Occasional bad frames can occur; the SDK handles them by restarting the stream if necessary.*

- *Performance and Threading: Processing every frame (especially at 25 Hz) can be CPU-intensive, particularly converting to bitmaps or running heavy computations. If the UI is lagging or frames are being dropped, ensure that the frame handling is off the main thread. The example above posts minimal work in the callback. The SDK sample uses a separate ImageThread for converting and drawing the image so that the USB thread isn't backed up [?] [?]. You should adopt a similar strategy: do the image conversion and any heavy analysis (e.g., running face detection on the thermal image) in a worker thread. This will prevent the frame queue from overflowing (which would manifest as increasing latency or stuttering).*

- *App Integration Issues: If integrating into an existing app like bucika*$_g sr, watchout for An$ $the user's USB permission is typically remembered only until the device is detached. If your app exits and the co$

- *Error Codes and Debugging: The SDK's ResultCode enum provides codes for failures (e.g., SUCCESS = 0, others for various errors). If uvcCamera.openUVCCamera(ctrlBlock) or irCmd.startPreview() returns non-zero, log or inspect those. Common errors might be related to device busy or invalid parameters. Ensure you are using the correct IRCMDType for your device model -- for TC001/Plus it's usually USB*$_I R_2 56_3 84 as$ $resolution" mentioned in marketing refers to upscaling/fusion, not an actual sensor pixel count). If you choo$

- *Image Orientation and Alignment: The camera's default orientation might not match your app's view. If you see the image rotated by 90° or mirrored, use*

the SDK's rotate() or mirror settings. The IRUVC helper class in the sample, for instance, has a setRotate(true) option which rotates the image 90° right [?]. You can also manually rotate the Bitmap before displaying. For alignment (especially for TC001 Plus fusion of visual and IR), ensure you use the provided alignment method from the SDK so that the thermal and visible images overlap correctly. If the visible image is not needed, you can ignore it; but if you do use it (say, to locate a face), remember the IR and RGB frames might have slight time offsets -- syncing them can be done by timestamp or by using the provided frame callback that perhaps delivers both in one call (depending on SDK capabilities).

If problems persist, consult the official Topdon SDK documentation (a PDF is provided in the SDK package) and the community forums. The Topdon community site has QAs -- for example, guidance for using the camera in custom apps [?] -- and the official FAQ addresses issues like "camera not recognized by phone" (often solved by ensuring the phone has OTG support or using the correct app) [?]. By systematically addressing the above points, you should be able to reliably use the TC001/TC001 Plus in your Android project and collect high-quality thermal data for your thesis work.

1.10 References

- Topdon TC001 Product Page -- Thermal Imaging Camera for Android Devices, 256×192 Resolution, -4°F to 1022°F range *[?] [?].*
- *Topdon TC001 Plus Specifications* -- Dual-Lens 256×192 IR Camera, 25 Hz frame rate, image fusion, ±2°C accuracy *[?] [?].*
- *Topdon Technology Thermal SDK (v1.3.7) -- Android Development Document, Nov 2023. (Includes API reference and sample code for image acquisition, pseudocolor, and temperature extraction). [?] [?]*
- *Gioia, F.* et al. *(2022). "Towards a Contactless Stress Classification Using Thermal Imaging."* Sensors, 22