1 Notice

Please read this document carefully. It contains what is provided in the SDK and how to use it, as well as some precautions and FAQs. It helps developers get familiar with the SDK quickly. If you have any question, please contact the technical support.

2 Provide files

This section mainly describes what is provided in the SDK and how to use each part of it. It helps developers get started quickly. Please read it carefully.

2.1 libirxxx_sdk_release

libxxx.aar

This file is the AAR package corresponding to the SDK.

For how to use it, please refer to `libir_sample`.

2.2 libirxxx_sample

This file is the reference demo of the SDK. The demo demonstrates the APIs and is for reference only.

Note: This directory contains a `README.md` file which details the image output process, instructions, precautions, and analysis of common issues. Please read it carefully.

2.3 libirxxx_apk

This file is the APK corresponding to the SDK's reference demo.

You can install this APK on your device for quick experience and test. You can also use it as a contrast for troubleshooting some issues arising in development.

2.4 LUT for user reference

This file is the table of temperature measurement distance corrections at high gain and low gain. Different tables may be used for different modules. The table here and the table under

`assets/tau` of `libir_sample` are both examples. For the specific table that you'll use, please contact the corresponding personnel to confirm.

2.5 API development documentation

This file is the documentation for API development on Android devices.

It mainly explains the tool classes, APIs and some complex functions provided in the SDK. Please read it carefully.

2.6 doc

This file describes the APIs of the libraries.

For comments on the APIs, request parameters and their types, as well as return values, please refer to this file.

Note: If you have any question about how to use a function during development, please query in this file.

2.7 User calibration instructions

This file explains lid-pattern noise calibration, secondary calibration, dead pixel calibration and ambient variable correction for user development.

2.7.1 Temperature measurement & lid-pattern noise calibration

This file explains temperature measurement and lid-pattern noise calibration in detail. It covers single-point calibration, two-point calibration, and multi-point calibration regarding the calibration in temperature measurement. `libxxx_sample` contains the corresponding demo examples.

2.7.2 Ambient variable correction

This file explains ambient variable correction in detail. It covers the function of temperature measurement correction. `libxxx_sample` contains the corresponding demo examples.

2.7.3 Dead pixel correction

This explains adding and removing dead pixels in detail. `libxxx_sample` contains the corresponding demo examples.

2.8 Tiny1c Android SPI Bring Up

This is the debugging file related to SPI.

This file will only be provided in the SDK of the SPI protocol.

3 SDK introduction

The Falcon Android SDK (hereinafter referred to as "SDK") implements the APIs for Falcon infrared cameras on Android. It provides the customer with the reference demo and low-level packaging libraries for secondary development.

4 Importing the SDK

1. Place the AAR libraries under the project directory.
2. Add a library directory under `build.gradle`.

```
dependencies {
    // use jar or aar
    implementation fileTree(dir: 'libs', include: ['*.jar','*.aar'])
    ...
    }
```

5 APIs

The JAVA layer of the SDK provide Java class libraries: `IRCMDType`, `UVCType`, `LibIRTemp`, `LibIRParse`, `LibIRProcess`, `IRCMD`, `UVCCAMERA`, `DualUVCCamera`, `IRUtils`, `CommonUtils`, `CommonParams`, and `USBMonitor`.

The role of each library is given below.

Note: The APIs listed here are only briefly explained to let users get a general understanding of the structure of the SDK. Different SDKs may use different APIs. For the request parameters, return values and detailed comments of a specific API, please refer to the documentation in `doc`.

5.1 IRCMDType

It is about movement command types. There are two types: USB and SPI.

It will be used when ICRMD is initialized. Please confirm the type of SDK that you want to use and the corresponding resolution. Different types of SDKs may have different APIs and parameters, and they cannot be mixed up.

5.2 IRCMD

It communicates with the movement firmware to enable information reading and writing, configuration functions, firmware upgrading, etc.

The SDK is released in the AAR format.

5.2.1 Initialization

```
..

// IRCMD init
ConcreteIRCMDBuilder concreteIRCMDBuilder = new ConcreteIRCMDBuilder();
            ircmd = concreteIRCMDBuilder
                    .setIrcmdType(IRCMDType.xxx)
                    .setIdCamera(uvcCamera.getNativePtr())
                    .build();
```

This is the standard way of SDK initialization. If you need bidirectional calibration, please refer to the document below to bind the SDK and the firmware bidirectionally.

5.2.2 APIs

For specific APIs and return values, please refer to the documentation in `doc`.

5.3 UVCType

It is about the UVC types. They are divided by protocol type.

It will be used when UVCCamera is initialized. Please confirm the protocol type in the SDK you are using. Different protocols cannot be mixed up.

5.4 UVCCamera

It is used to search for UVC camera equipment, monitor the plug and unplug of the equipment, open the connection and call back the picture, and send the corresponding control commands.

5.4.1 Initialization

```
..

// UVCCamera init
ConcreateUVCBuilder concreateUVCBuilder = new ConcreateUVCBuilder();
uvcCamera = concreateUVCBuilder
            .setUVCType(UVCType.xxx)
            .bind();
```

By setting different values for cameraWidth and cameraHeight, you can control the resolution of different pictures.

```
uvcCamera.setUSBPreviewSize(cameraWidth, cameraHeight);
```

- cameraWidth: 256, cameraHeight: 384, infrared + temperature
- cameraWidth: 256, cameraHeight: 192, infrared by default

  You can get different data flow in the Y16 mode by calling the startY16ModePreview API is called and transferring different parameters.

5.4.2 APIs

For specific APIs and return values, please refer to the documentation in `doc`.

5.5 DualUVCCamera

It will be used by the dual-light SDK only.

5.5.1 Initialization

```
..

// DualUVCCamera initialization
ConcreateDualBuilder concreateDualBuilder = new ConcreateDualBuilder();
dualUVCCamera = concreateDualBuilder

                .setDualType(DualType.xxx)
                .setIRSize(xxx, xxx)
                .setVLSize(xxx, xxx)
                .setDualSize(xxx, xxx)
                .setDataFlowMode(dataFlowMode)
                .setPreviewCameraStyle(CommonParams.PreviewCameraStyle.xxx)
                .setDeviceStyle(CommonParams.DeviceStyle.xxx)
                .build();
```

5.5.2 APIs

For specific APIs and return values, please refer to the documentation in `doc`.

5.6 LibIRTemp

It is used to parse and process temperature data, and perform secondary correction over the temperature data.

5.6.1 APIs

For specific APIs and return values, please refer to the documentation in `doc`.

5.7 LibIRParse

It is used to parse raw data into temperature and image, and convert data formats, such as conversion between RGB and YUV.

5.7.1 APIs

For specific APIs and return values, please refer to the documentation in `doc`.

5.8 LibIRProcess

It is about some additional processing algorithms, such as pseudo-coloring, rotation, mirroring, flip, etc.

5.8.1 APIs

For specific APIs and return values, please refer to the documentation in `doc`.

5.9 IRUtils

It is about infrared-related APIs used for operations when the device is offline. They are suitable for scenarios such as secondary editing of pictures.

5.10 CommonUtils

It provides some public functions for convenience of developers.

5.11 CommonParams

It is about the public parameter class of the function. It gives restriction to parameter types to avoid errors.

5.12 USBMonitor

It is used to monitor the plugging and unplugging of USB devices. The SPI protocol does not need it.

5.12.1 APIs

For specific APIs and return values, please refer to the documentation in `doc`.

6 APP Development Reference Guide

6.1 Introduction to the processes of image output and temperature measurement of the app

- USBcamera handles the class IRUVC to realize the monitoring of camera insertion status, the control of USB permissions, the initialization of camera parameters, and the callback of camera connection preview instruction and data retrieving. Open the camera in the callback of onConnect, and realize the retrieving, copying, splitting and rotation of image data and temperature data in the callback function onFrame.

- The image data is called back and sent to ImageThread for processing and conversion to produce a file in the ARGB format. Then CameraView uses bitmap and enlarges the ARGB-formatted data to fill the screen for imaging drawing.
- TemperatureView realizes the display of temperature measurements by dots, lines, and planes. The onTouch method allows drawing dots, lines, and planes with fingers and recording their coordinates. The process of temperature measurement calls the temperature measurement library to measure temperatures and finally displays them at the corresponding coordinates.
- The infrared movement firmware provides the API for the configuration of image and testing parameters, which can improve the quality of images and temperature measurement. The class Libircmd provides a series of functions to call. The camera needs to be turned on so that the camera handle pointer can be called.

6.2 Device pid filter

The UVC chip pid has a default value. If you device's pid is not the same as the default value, you need to change the actual filter parameters.

6.3 Temperature measurement function

6.3.1 Temperature measurement with the Libirtemp library

- Calculation of RAW temperature data into Celsius:

  x and y are coordinate data. cameraWidth is the pixel width of the camera. If the data is rotated, then the rotated width will apply.

```
(temperature[(x*cameraWidth+y)*2]+((int)(temperature[(x*cameraWidth+y)*2+1])
<<8))/64 -273.15;
```

- The temperature measurement library provides the APIs for temperature measurement by dots, lines, and planes.

Call the setTempData method of Libirtemp and transfer the temperature data. Then you can call the corresponding method to get the temperature information.

```
/**

    * copy Temperature data from buffer<br/>
    *
    * @param src Temperature buffer
    */
    public void setTempData(byte[] src)
```

- Return the temperature at the coordinate dot in Celsius.

```
Libirtemp.getTemperatureOfPoint(Point point)
```

- Return the lowest, highest, and average temperatures within the line, and the corresponding coordinates to the lowest and highest temperatures.

```
Libirtemp.getTemperatureOfLine(Line line)
```

- Return the lowest, highest, and average temperatures within the plane, and the corresponding coordinates to the lowest and highest temperatures.

```
Libirtemp.getTemperatureOfRect(Rect rect)
```

6.3.2 Temperature measurement with the IRCMD library

- Return the temperature of the coordinate point in Celsius

```
public int getPointTemperatureInfo(int pixelPointX, int pixelPointY, int[]
temperatureValue)
```

- Return the lowest, highest, and average temperatures within the line, and the corresponding coordinates to the lowest and highest temperatures.

```
public int getLineTemperatureInfo(int startPointX, int startPointY, int
endPointX, int endPointY, int[] temperatureValue)
```

- Return the lowest, highest, and average temperatures within the plane, and the corresponding coordinates to the lowest and highest temperatures.

```
public int getRectTemperatureInfo(int startPointX, int startPointY, int
endPointX, int endPointY,
                                        int[] temperatureValue)
```

- Get the minimum temperature of the frame.

```
public int getCurrentFrameMinTemperature(int[] temperatureValue)
```

- Get the maximum temperature of the frame.

```
public int getCurrentFrameMaxTemperature(int[] temperatureValue)
```

- Get the information of the maximum and minimum temperatures of the frame.

```
public int getCurrentFrameMaxAndMinTemperature(int[] temperatureValue)
```

6.4 Secondary correction in temperature measurement

For details, please refer to the file: `User calibration instructions\Ambient variable correction\Ambient variable correction.pdf`.

6.5 Automatic switching between high gain and low gain

```
...
private LibIRProcess.AutoGainSwitchInfo_t auto_gain_switch_info = new
LibIRProcess.AutoGainSwitchInfo_t();
Private LibIRProcess.GainSwitchParam_t gain_switch_param = new
LibIRProcess.GainSwitchParam_t();
...

        // auto gain switch parameter
        gain_switch_param.above_pixel_prop = 0.1f;    //To switch high gain to low
gain; percentage of the total area of the device pixels
        gain_switch_param.above_temp_data = (int)((130 + 273.15) * 16 * 4); //To
switch high gain to low gain; the temperature to trigger the switching
        gain_switch_param.below_pixel_prop = 0.95f;    //To switch low gain to high
gain; the percentage of the total area of the device pixels
        gain_switch_param.below_temp_data = (int)((110 + 273.15) * 16 * 4); //To
switch low gain to high gain; the temperature to trigger the switching
        auto_gain_switch_info.switch_frame_cnt = 5 * 15; //The automatic gain
switching will be triggered when the number of frames that constantly meet the
conditions exceeds this threshold (If the speed of image output is 15 frames per
second, then it will be approximately 5 seconds.)
        auto_gain_switch_info.waiting_frame_cnt = 7 * 15;//After the automatic gain
switching is triggered, the frames at this interval will not be monitored in the
switching of gain conversion (If the speed of image output is 15 frames per second,
then it will be approximately 7 seconds.)

...

/**
    * Automatic switching between high gain and low gain<br/>
    * Each frame is processed.<br/>
    *
    * @param temp_frame               Original temperature data<br/>
    * @param image_res                Width and height of the original temperature
data<br/>
    * @param auto_gain_switch_info    The frame threshold of triggering automatic gain
switching. When it is monitored that the number of frames with high gain or low gain
under the gain_switch_param condition exceeds that under the auto_gain_switch_info
condition, the automatic gain switching will be triggered.<br/>
    * @param gain_switch_param        Parameters for automatic gain switching
    * @param autoGainSwitchCallback   Callback of automatic switching
    * @return see {@link IrcmdResult}
    */
   Public int autoGainSwitch(byte[] temp_frame, LibIRProcess.ImageRes_t image_res,
                             LibIRProcess.AutoGainSwitchInfo_t auto_gain_switch_info,
                             LibIRProcess.GainSwitchParam_t gain_switch_param,
                             AutoGainSwitchCallback autoGainSwitchCallback)
```

The calling method can call each frame in the callback IRUVC onframe.

6.6 Overexposure protection

```
...
// over_protect parameter
        int low_gain_over_temp_data = (int)((550 + 273.15) * 16 * 4); //The
temperature to trigger protection against overexposure at low gain
        int high_gain_over_temp_data = (int)((150 + 273.15) * 16 * 4); //The
temperature to trigger protection against overexposure at high gain
        float pixel_above_prop = 0.02f; //The percentage of the total area of the
device pixels
        int switch_frame_cnt = 7 * 15; //If the triggering condition constantly
exceeds the threshold, it will trigger protection against overexposure. (In case that
the speed of image output is 15 frames per second, then it will be approximately 7
seconds.)
        int close_frame_cnt = 10 * 15; //After protection against overexposure is
triggered, the number of frames that exceeds the threshold will then terminate
protection against overexposure. (In case that the speed of image output is 15 frames
per second, then it will be approximately 10 seconds.)
...
/**
    * Overexposure protection <br/>
    * Each frame is processed. <br/>
    *
    * @param isUseIRISP                    Whether to use the ISP algorithm; false in
general
    * @param gainStatus                    The current gain status {@link
CommonParams.GainStatus#HIGH_GAIN}{@link CommonParams.GainStatus#LOW_GAIN}<br/>
    * @param temp_frame                    The original temperature data<br/>
    * @param image_res                    The width and height of the original
temperature data<br/>
    * @param low_gain_over_temp_data    The temperature value to trigger protection
against overexposure at low gain<br/>
    * @param high_gain_over_temp_data    The temperature value to trigger protection
against overexposure at high gain<br/>
    * @param pixel_above_prop            When the portion of the pixels in a frame
exceeds the threshold, it will trigger protection against overexposure.<br/>
    * @param switch_frame_cnt            The number of frames that constantly trigger
protection against overexposure<br/>
    * @param close_frame_cnt            After protection against overexposure is
triggered, the shutter will open when the number of frames exceeds the threshold.<br/>
    * @param avoidOverexposureCallback    Callback of avoiding overexposure, true:
triggering protection against overexposure; false: terminating protection against
overexposure
    * @return see {@link IrcmdResult}
    */
   public int avoidOverexposure(boolean isUseIRISP, CommonParams.GainStatus
gainStatus, byte[] temp_frame,    LibIRProcess.ImageRes_t image_res, int
low_gain_over_temp_data, int high_gain_over_temp_data, float pixel_above_prop,
int switch_frame_cnt, int close_frame_cnt, AvoidOverexposureCallback
avoidOverexposureCallback)
```

The calling method can call each frame in the callback IRUVC onframe.

6.7 Different data flows

Switching to different modes of image output will output data flows with different resolutions and in different modes.

dataFlow	data	resolution
IMAGE_AND_TEMP_OUTPUT	Image + Temp	256*384
IMAGE_OUTPUT	Image	256*192
TEMP_OUTPUT	Temp(Toggle via startY16ModePreview)	256*192

At the time of `UVCCamera` intialization, transferring different resolutions will switch to different modes of data flows:

```
uvcCamera = new UVCCamera(cameraWidth, cameraHeight);
/**
         * cameraWidth:256,cameraHeight:384,image+temp
         * cameraWidth:256,cameraHeight:192,image
         * cameraWidth:256,cameraHeight:192,(call startY16ModePreview,
transferY16_MODE_TEMPERATURE) temp
         */
```

6.8 Custom pseudo-color

6.8.1 Introduction

For the convenience of customers in using pseudo-color tables defined by themselves, APIs and examples are provided for generation of pseudo-color tables, format conversion, and data storage and reading.

6.8.2 Generating custom pseudo-color tables

```
                // Generating a custom pseudo-color table
                int[][] color1 = new int[][]{{0}, {0, 0, 40}};
                int[][] color2 = new int[][]{{21}, {138, 20, 150}};
                int[][] color3 = new int[][]{{42}, {248, 135, 0}};
                int[][] color4 = new int[][]{{209}, {208, 48, 75}};
                int[][] color5 = new int[][]{{230}, {249, 143, 0}};
                int[][] color6 = new int[][]{{255}, {255, 255, 196}};
                byte[] pseudoDataByte =
CommonUtils.generatePseudocolorData(color1, color2, color3, color4, color5, color6);
```

`color1, color2, color3, color4, color5, color6` are sampling points. At least two of them are required. The more, the better.

6.8.3 Format conversion for pseudo-color tables

The format of a generated pseudo-color table can be converted for different scenarios.

- Converting into RGB

```
CommonUtils.convertRGBPseudocolorData
```

- Converting into YUV

```
CommonUtils.convertYUVPseudocolorData
```

6.8.4 Applying custom pseudo-color

By default, the system pseudo-color will be applied. Transfer pseudo-color types.

```
LibIRProcess.convertYuyvMapToARGBPseudocolor
```

Apply custom pseudo-color, and transfer data from pseudo-color tables.

```
LibIRProcess.convertYuyvMapToARGBCustomPseudocolor
```

6.8.5 Steps to customize pseudo-coloring for USB dual views

The custom pseudo-coloring for USB dual views is different.

6.8.5.1 Generating a custom pseudo-coloring table

```
// Generating a custom pseudo-color table
                int[][] color1 = new int[][]{{0}, {0, 0, 40}};
                int[][] color2 = new int[][]{{21}, {138, 20, 150}};
                int[][] color3 = new int[][]{{42}, {248, 135, 0}};
                int[][] color4 = new int[][]{{209}, {208, 48, 75}};
                int[][] color5 = new int[][]{{230}, {249, 143, 0}};
                int[][] color6 = new int[][]{{255}, {255, 255, 196}};
                byte[] pseudoDataByte =
CommonUtils.generatePseudocolorData(color1, color2, color3, color4, color5, color6);
```

`color1, color2, color3, color4, color5, color6` are sampling points. At least two of them are required. The more, the better.

6.8.5.2 Rendering pseudo-coloring

Call the loadCustomPseudocolor and setCustomPseudocolor methods. The name of pseudo-color can be self-defined.

```
dualView.getDualUVCCamera().loadCustomPseudocolor("custom", data);
dualView.getDualUVCCamera().setCustomPseudocolor("custom");
```