Camera

Programming Guide

Version 1.3.2

Table of Contents

1.Overview

Camera SDK provides enhanced features and APIs available to users for developing enriched camera-based applications. This SDK offers additional functionalities and capabilities on top of existing Android Camera API2 capabilities. The Camera application interface utilizes the Android Camera APIs. Camera SDK also provides easy-to-use, fast performing APIs for new image processing and computational photography needs.

Camera SDK Processors provide the user with features such as High Dynamic Range, Low-Light environment photography, Depth of Field, and Landscape photography (Panorama).

The Camera SDK can be used to:

- Develop camera applications which uses new features added on top of Android APIs
- Add image processing features to the application to provide delightful capabilities to the camera application

1.1 Basic Knowledge

The **`com.samsung.android.sdk.camera`** package provides an interface to the **`android.hardware.camera2`** package. It can access individual camera devices connected to a Samsung Android device, provide Samsung specific hardware controls like Phase Auto-Focus, Metering Mode, Real Time HDR, etc., and present additional features and utilities which can be used to develop sophisticated camera based or image processing applications.

For more information, please refer to the android.hardware.camera2 documentation.

1.2 Architecture

The architecture consists of:

- **Applications**: Applications that use the Camera SDK
- **Camera API**: API for camera functionality, applying filters and various processors like HDR, low light, Panorama, etc
- **Android Camera API2**: Android Camera API2 camera components.
- **Native Engine:** Engine for native side implementation

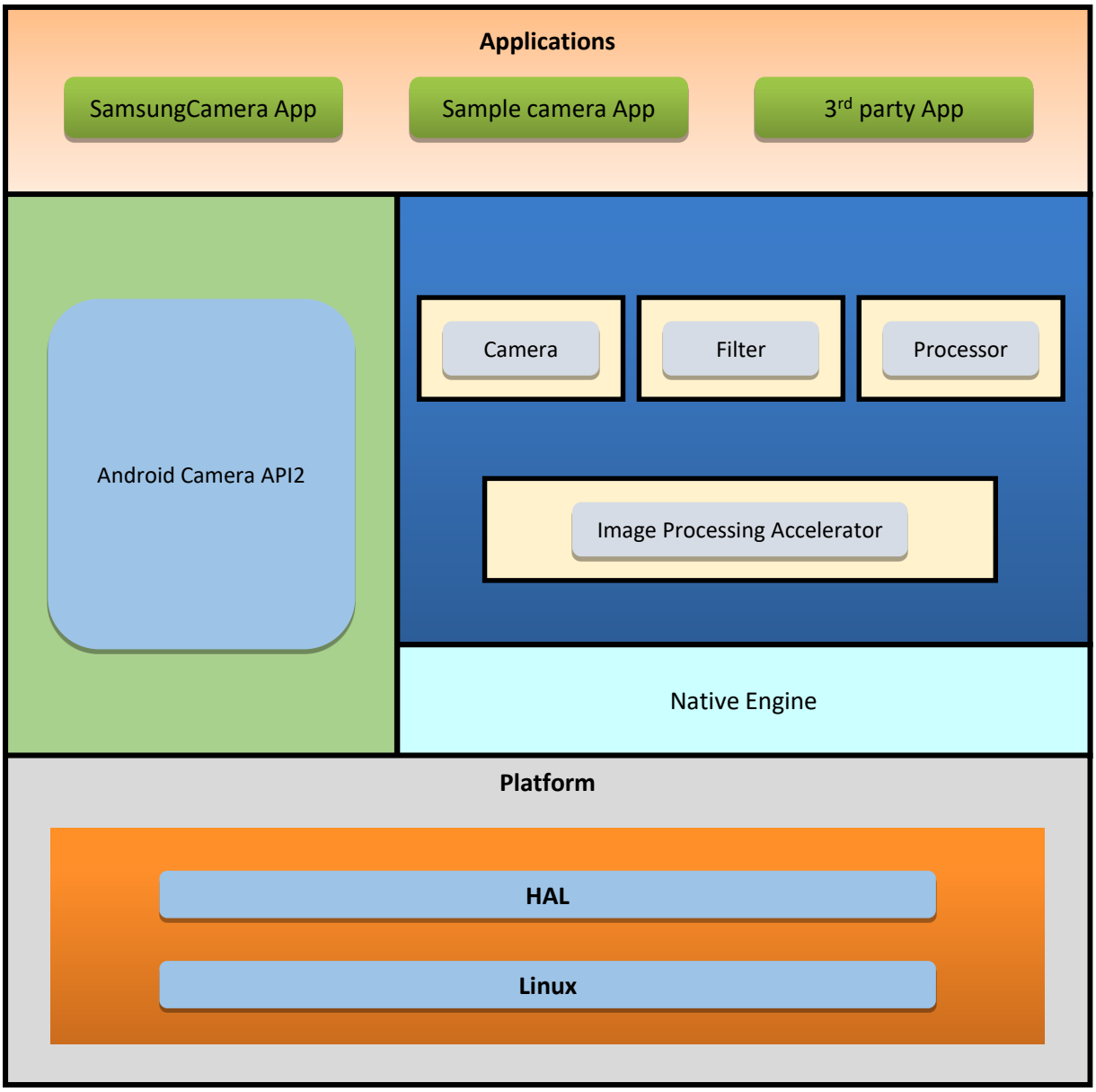

Figure 1: Camera Packages and Classes

1.3 Package Diagram

Figure 2 shows the Camera packages and classes that can be used in the application.

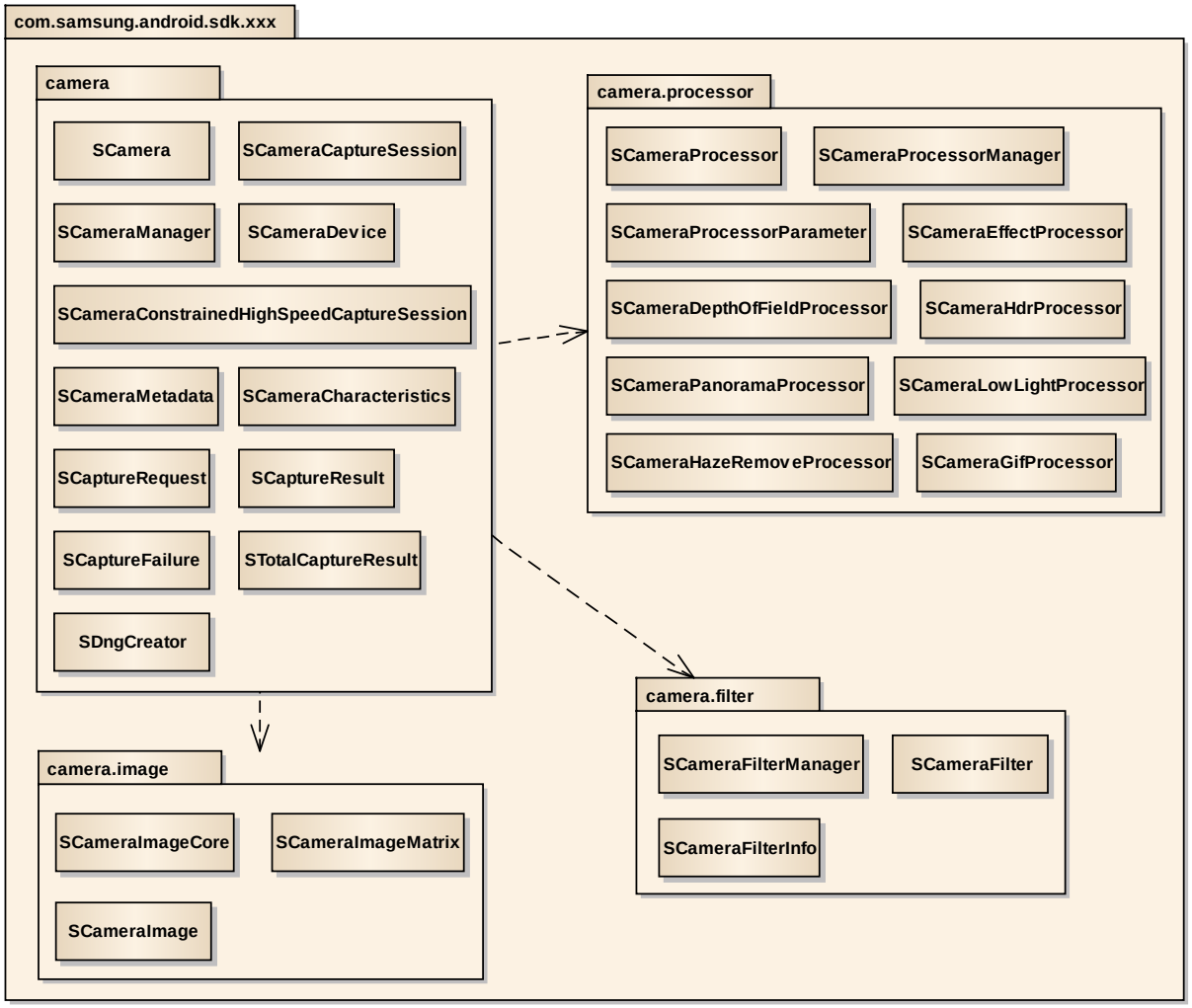

Figure 2: Camera Packages and Classes

The Camera packages and classes include:

- **SCamera**: Provides version and feature capability check APIs
- **SCameraManager**: A system service manager for detection, characterization, and connection of the SCameraDevice
- **SCameraDevice**: Represents a single camera connected to an Android device
- **SCameraCaptureSession**: A configured capture session for a SCameraDevice
- **SCameraConstrainedHighSpeedCaptureSession**: A constrained high speed capture session for a SCameraDevice, used for capturing high speed images
- **SCameraMetadata**: Base class for camera controls and information

- **SCameraCharacteristics**: Properties describing an SCameraDevice
- **SCaptureRequest**: An immutable package of settings and outputs needed to capture a single image
- **SCaptureResult**: Subset of the results of a single image capture from the image sensor
- **STotalCaptureResult**: Total assembled results of a single image capture from the image sensor
- **SCaptureFailure**: A report of failed capture for a single image capture
- **SDngCreator**: Provides functions to write raw pixel data as a DNG file
- **SCameraProcessor**: Base class for all processors
- **SCameraProcessorManager**: Manages the processors available in the device
- **SCameraProcessorParameter**: Defines the key/value set related to a processor
- **SCameraHdrProcessor**: Processor to apply HDR image processing on a list of images
- **SCameraEffectProcessor**: Applies the filter effect on image data
- **SCameraDepthOfFieldProcessor**: Processor to apply depth of field effect on a list of images
- **SCameraLowLightProcessor**: Processor to process the images which are captured under a low light environment or a dark and shim environment
- **SCameraPanoramaProcessor**: Processor for getting Panorama images
- **SCameraHazeRemoveProcessor**: Processor to apply haze remove effect on the image data
- **SCameraGifProcessor**: Processor that aggregates series of image data and save to animated GIF
- **SCameraFilterManager**: Manages creation and retrieval of available filters
- **SCameraFilter**: Contains the information of each filter
- **SCameraFilterInfo**: Represents information of each filter
- **SCameraImage**: Image buffer with high quality image size to be used with Image package
- **SCameraImageCore**: Framework for image processing
- **SCameraImageMatrix**: Support for matrix functionality

Replace xxx in the image above with camera, camera.processor, camera.filter or camera.image to get the full package name in question.

1.4 Supported Platforms

Android 5.0 (Lollipop) or above supports the Camera SDK.

1.5 Supported Features

SCamera supports the following features:

- **Camera:** This package provides basic functionalities that are required for camera initialization, capture, recording, etc. It allows access to individual camera devices connected to a Samsung Android device. It also allows management of Samsung specific hardware controls like Phase Auto-Focus, Metering Mode, Real Time HDR, etc.

- **Image:** Image is a framework which works in a heterogeneous environment. This allows the application developers to acquire maximum benefit in terms of performance. This framework supports image processing operations and mathematical operations.

- **Processor:** This package offers various camera add-on functionalities like HDR, Low Light, and Depth of Field capture, thereby enabling the users to capture in different modes.

- **Filter:** This package is used in conjunction with `SCameraEffectProcessor` to let users apply various image filtering effects on captured images as well as for stream processing. Third party developers can also develop filters that can be downloaded.

1.6 Components

- Components
    - camera-v1.3.2.aar
    - camera-v1.3.2-light.aar
    - sdk-v1.0.0.jar
- Imported package
    - com.samsung.android.sdk.camera

1.7 Importing Libraries

From the Camera SDK v1.2.2, the Light Packaging is introduced to reduce the APK footprint. While the Light Packaging only supports **`SCAMERA_FILTER`** feature, `SCameraEffectProcessor` from **Processor** package and the interface to the **`android.hardware.camera2`** package, it reduces about 18MB of APK footprint.

To import the Camera SDK to the application project:

1. Add **camera-v1.3.2.aar** and **sdk-v1.0.0.jar** to the libs folder to support all features of Camera SDK (Figure 3).

2. To use Light Packaging, add **camera-v1.3.2-light.aar** instead of **camera-v1.3.2.aar** to the libs folder (Figure 4).

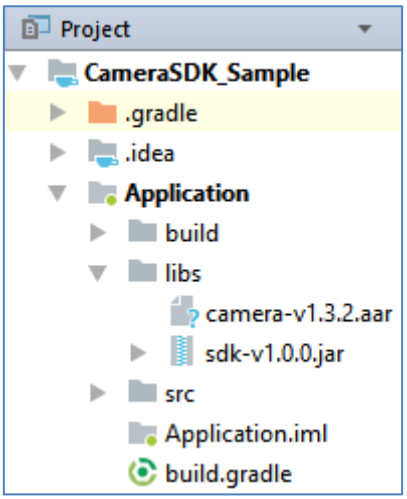

Figure 3: libs folder in Android Studio

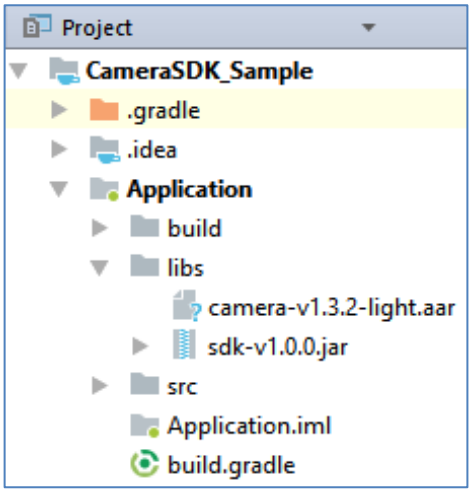

Figure 4: libs folder in Android Studio for Light Packaging

3. Add the following lines to your build.gradle

```
repositories {
  flatDir {
```



```
    dirs 'libs'
  }
}

dependencies {
  // before gradle 3.4
  compile(name: 'camera-v1.3.2-light', ext: 'aar') // for full package
  compile(name: 'camera-v1.3.2-light', ext: 'aar') // for light package
  compile files('libs/sdk-v1.0.0.jar')

  // for gradle 3.4 or later
  implementation(name: 'camera-v1.3.2-light', ext: 'aar') // full pkg.
  implementation(name: 'camera-v1.3.2-light', ext: 'aar') // light pkg.
  implementation files('libs/sdk-v1.0.0.jar')
}
```

4. Add the following permission to the application manifest if the application needs Camera access:

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

5. Select **Android 5.0 or higher** as a project build target in the project properties.

6. Following permissions are merged into AndroidManifest.xml during build process to initialize the SDK and to use filters, Face AR sticker and AR Emoji sticker:

```
<uses-permission android:name=
"com.samsung.android.provider.filterprovider.permission.READ_FILTER" />
<uses-permission android:name=
"com.samsung.android.provider.stickerprovider.permission.READ_STICKER_P
ROVIDER" />
<uses-permission android:name=
"com.samsung.android.aremoji.provider.permission.READ_STICKER_PROVIDER"
/>
<uses-permission android:name=
"com.samsung.android.providers.context.permission.WRITE_USE_APP_FEATURE
_SURVEY" />
```


2. Using the SCamera class

The following code shows how to use the SCamera class methods:

```
mScamera = new SCamera();
try {
    mScamera.initialize(this);
    return;

} catch (SsdkUnsupportedException e) {
    if (eType == SsdkUnsupportedException.VENDOR_NOT_SUPPORTED) {
        // The device is not a Samsung device.
    } else if (eType == SsdkUnsupportedException.DEVICE_NOT_SUPPORTED) {
        // The device does not support Camera.
    } else if (eType == SsdkUnsupportedException.SDK_VERSION_MISMATCH) {
        // There is a SDK version mismatch.
    }
}
mSCameraManager = mScamera.getSCameraManager();
int versionCode = mScamera.getVersionCode();
String versionName = mScamera.getVersionName();
```

The SCamera class provides the following methods:

- **`initialize(Context)`** - Initializes an instance of the Camera SDK package. SCamera needs to be initialized before use. If SDK is not supported on the device, **SsdkUnsupportedException** is thrown
- **`getSCameraManager()`** - Returns an instance of SCameraManager for detecting, characterizing, and connecting to SCameraDevice
- **`getVersionCode()`** - Returns the version code of the Camera SDK package
- **`getVersionName()`** - Returns the version name of the Camera SDK package
- **`isFeatureEnabled(int type)`** - Checks whether the requested feature is supported or not. The type parameter is the identifier of the feature which can either be SCAMERA_FILTER, SCAMERA_PROCESSOR, or SCAMERA_IMAGE.

2.1 Using the initialize() method

The `SCamera` **`initialize()`** method:

- initializes Camera
- loads the Native Engine of SCameraSDK

- checks if the Samsung device supports Camera

If the initialization of `SCamera` failed, the **`initialize()`** method throws an **SsdkUnsupportedException** exception. To find out the reason for this exception, check the exception message.

2.2 Handling SsdkUnsupportedException

If an **`SsdkUnsupportedException`** exception is thrown, check the exception message type using **`SsdkUnsupportedException.getType()`.**

The following types of exception messages needs to be handled:

- **`DEVICE_NOT_SUPPORTED`:** The device does not support SCamera
- **`SDK_VERSION_MISMATCH`:** The SDK version is mismatched
- **`VENDOR_NOT_SUPPORTED`:** The device is not a Samsung device

2.3 Checking the Availability of Camera Features

To verify if an `SCamera` feature is supported on the device, use the **`isFeatureEnabled()`** method. The feature types are defined in the `SCamera` class. Pass the feature type as a parameter when calling the **`isFeatureEnabled()`** method. The method returns a boolean value that indicates the support for the feature on the device.

```
boolean isFeatureEnabled(int type)
```

The following types of Camera features are defined in SCamera class:

- **`SCAMERA_PROCESSOR`:** A set of SCameraProcessor processor is available.
- **`SCAMERA_FILTER`:** A set of SCameraFilter filter is available
- **`SCAMERA_IMAGE`:** A set of image processing APIs

3. Using SCameraProcessor

The **SCameraProcessor** class allows the creation of a single image after processing multiple/single image, or processing over a continuous stream of data (e.g., Camera preview). SCameraProcessor, being the base class of all Processors, gives the user methods to interact with all processors in general. It defines the following common types of exception messages for error callback:

- **NATIVE_PROCESSOR_MSG_DECODING_FAIL:** indicates failure in jpeg decoding
- **NATIVE_PROCESSOR_MSG_ENCODING_FAIL:** indicates failure in jpeg encoding
- **NATIVE_PROCESSOR_MSG_PROCESSING_FAIL:** indicates failure in processing the processor
- **NATIVE_PROCESSOR_MSG_UNKNOWN_ERROR:** indicates an unknown error

It defines the following common fields which are used as a key for **SCameraProcessParameter** class and defines the guidelines for each SCameraProcessor objects for execution. Also, the Processor specific parameter keys (which are defined by the processor itself) are explained under the respective processor.

- **CAMERA_ID**: ID of camera device which is used to provide image data to processor
- **JPEG_QUALITY**: Jpeg quality
- **MULTI_INPUT_COUNT_RANGE:** Input count range for multi-input image processor
- **STILL_INPUT_FORMAT**: Still input format
- **STILL_INPUT_FORMAT_LIST:** List of available still input format
- **STILL_OUTPUT_FORMAT:** Still output format
- **STILL_OUTPUT_FORMAT_LIST**: List of available still output format
- **STILL_SIZE**: Size of still input
- **STILL_SIZE_LIST**: List of available still input size
- **STREAM_SIZE**: Stream input size
- **STREAM_SIZE_LIST**: List of available stream input size range

Each Processor requires an initial set of values to some of its keys. The **SCameraProcessorParameter** class is used to supply the values to these keys.

The **SCameraProcessorParameter** class is used to set and get the values of keys defined in **SCameraProcessor** for each processor. It has the following methods:

- **get(Key key):** Gets a parameter field value
- **set(Key key,T value):** Sets value to a parameter field

- **getAvailableKeys():** Lists the keys of supported parameter fields

SCameraProcessor has a following common interface which is implemented by each Processor.

- **initialize():** Initializes the processor
- **deinitialize():** Deinitializes SCameraProcessor
- **close():** Closes the SCameraProcessor
- **getParameters():** Returns the current settings for this processor
- **setParameters(SCameraProcessorParameter param):** Sets the changes settings to this Processor

To verify if the SCameraProcessor feature is supported on the device, use the **isFeatureEnabled()** method.

NOTE: The type value for **isFeatureEnabled(int type)** is **SCAMERA_PROCESSOR.**

Before using any Processor, provide the initial settings to it and make a call to the **initialize()** method**.** This method initializes the processor and prepares it for use. After using it, make a call to **deinitialize()**. This method frees the Processor resources held with it.

For the SDK version 1.1.0, the **close()** method is added as a common interface of the SCameraProcessor. Once the SCameraProcessor is closed, all methods in it will throw an IllegalStateException.

NOTE: The initialization keys cannot be changed after **initialize()** is called. To change key values, **deinitialize()** needs to be called first. Initialization keys for each processor are listed under the Initializing Processor section.

```
mProcessor = mSCameraProcessorManager.getProcessorInstance(
       SCameraProcessorManager.PROCESSOR_TYPE_PANORAMA);
SCameraProcessorParameter param = mProcessor.getParameters();
param.set(SCameraPanoramaProcessor.STILL_OUTPUT_FORMAT, ImageFormat.JPEG);
param.set(SCameraPanoramaProcessor.STREAM_SIZE, previewSize);
param.set(SCameraPanoramaProcessor.CAMERA_ID, Integer.parseInt(mCameraId));

mProcessor.setParameters(param);
mProcessor.initialize();
/* do something with SDK */
mProcessor.deinitialize();
```


SCameraProcessorManager manages the available processors and creates an instance of the processor on request. **SCameraProcessorManager** has the keys for accessing the available processors types. It has the following keys:

- **PROCESSOR_TYPE_EFFECT**: `SCameraProcessorManager.ProcessorType` indicates effect processor
- **PROCESSOR_TYPE_HDR**: `SCameraProcessorManager.ProcessorType` indicates high-dynamic range processor
- **PROCESSOR_TYPE_LOW_LIGHT:** `SCameraProcessorManager.ProcessorType` indicates light processor
- **PROCESSOR_TYPE_DEPTH_OF_FIELD:** `SCameraProcessorManager.ProcessorType` indicates depth of field processor.
- **PROCESSOR_TYPE_PANORAMA:** `SCameraProcessorManager.ProcessorType` indicates panorama processor

An **SCameraProcessorManager** instance can be acquired using an `SCamera` object. The developer can search and get the available processor types and create an instance of a required type.

NOTE: If a prior SCameraProcessor instance already exists when **createProcessor()** is called, the previous instance will be closed. The current SDK version does not support multiple instances of an SCameraProcessor.

```
mScamera = new SCamera();
try {
   mScamera.initialize(this);
} catch (SsdkUnsupportedException e) {
    //Exception handling for non support of SDK
    ...
}
/* APIs under com.samsung.android.sdk.camera.processor are only supported
 * when SCAMERA_PROCESSOR feature is enabled.
 */
if(mScamera.isFeatureEnabled(SCamera.SCAMERA_PROCESSOR)) {
    /* Gets SCameraProcessorManager instance */
    SCameraProcessorManager mSCameraProcessorManager =
    mScamera.getSCameraProcessorManager();

    /* Availability of each processor can be queried */
    if(mSCameraProcessorManager.isProcessorAvailable(
         SCameraProcessorManager.PROCESSOR_TYPE_LOW_LIGHT)) {

    SCameraLowLightProcessor mProcessor =
```



```
      mSCameraProcessorManager.createProcessor (
         SCameraProcessorManager.PROCESSOR_TYPE_LOW_LIGHT);
       /* do something with processor */
       ...
    }

    /* or whole list of supported processor in this device can be obtained */
    List<SCameraProcessorManager.ProcessorType<?>> processors =
     mSCameraProcessorManager.getAvailableProcessorTypeList();
    if(processors.contains(SCameraProcessorManager.PROCESSOR_TYPE_LOW_LIGHT)) {
        SCameraLowLightProcessor mProcessor =
         mSCameraProcessorManager.getProcessorInstance(
             SCameraProcessorManager.PROCESSOR_TYPE_LOW_LIGHT);
       /* do something with processor */
       ...
    }
}
```

3.1 SCameraHdrProcessor

SCameraHdrProcessor creates a high-definition image from a set of 3 images captured with different exposure values. This class inherits from `SCameraProcessor`.

3.1.1 Initializing SCameraHdrProcessor

Processor is initiated from `ProcessorManager`. The following `SCameraProcessorParameter` keys need values to initialize the `SCameraHdrProcessor`:

- **STILL_SIZE** : Still image size
- **STILL_INPUT_FORMAT:** Still image input format
- **STILL_OUTPUT_FORMAT:** Still image output format

```
SCameraProcessorManager processorManager =
mSCamera.getSCameraProcessorManager();

if(!processorManager.isProcessorAvailable
    (SCameraProcessorManager.PROCESSOR_TYPE_HDR)) {
    //This device does not support HDR Processor.

    return false;
}
```



```
mProcessor = processorManager.createProcessor
    (SCameraProcessorManager.PROCESSOR_TYPE_HDR);
SCameraProcessorParameter param = mProcessor.getParameters();

param.set(SCameraHdrProcessor.STILL_INPUT_FORMAT, ImageFormat.JPEG);
param.set(SCameraHdrProcessor.STILL_OUTPUT_FORMAT, ImageFormat.JPEG);
param.set(SCameraHdrProcessor.STILL_SIZE, new Size(mImageReader.getWidth(),
mImageReader.getHeight()));

mProcessor.setParameters(parameter);
mProcessor.initialize();
```

The callbacks from the `SCameraHdrProcessor` originate from the `SCameraHdrProcessor` through the `SCameraHdrProcessor.EventCallback` interface. You need to implement the required methods of `EventCallback` Interface. It has the following methods:

- **`onError(int code)`:** Called when an error occurs.
- **`onProcessCompleted(Image result)`:** Called when the `requestMultiProcess(List)` is done

Error conditions can be handled in `onError`, and after process completion, we get the final image.

```
mProcessor.setEventCallback(new SCameraHdrProcessor.EventCallback() {
    @Override
    public void onProcessCompleted(Image result) {
        //Result image handling needs to be done here

    }

    @Override
    public void onError(int code) {
        //Error code handling

    }
}, mBackgroundHandler);
```

3.1.2 Providing Input

Three images with different exposure values are captured and then supplied to the `requestMultiProcess()` method of `SCameraHdrProcessor`. The exposure values of all three

images should lie between the supported exposure range. For example, if exposure value range is [-1, 1], then three image of exposure -1, 0, 1 should be captured.

```
MAX_COUNT =
    param.get(SCameraHdrProcessor.MULTI_INPUT_COUNT_RANGE).getUpper();
mImageList = new ArrayList<Image>();
mJpegReader = ImageReader.newInstance(jpegsize.getWidth(), jpegsize.getHeight(),
     ImageFormat.JPEG, MAX_COUNT + 1);
mJpegReader.setOnImageAvailableListener(new
     ImageReader.OnImageAvailableListener() {
    @Override
    public void onImageAvailable(ImageReader reader) {
        CUR_COUNT++;
        showTextToDebugMessageView("Input image captured(" + CUR_COUNT + "/" +
            MAX_COUNT + ")");

        mImageList.add(reader.acquireNextImage());

        if(CUR_COUNT == MAX_COUNT) {
            mProcessor.requestMultiProcess(mImageList);
            int d = 0;
            for(Image i : mImageList) {
                mImageSaver.save(i, "Hdr_INPUT(" + ++d + ").jpg");
            }
            mImageList.clear();
        }
    }
}, mBackgroundHandler);
```

3.2 SCameraLowLightProcessor

SCameraLowLightProcessor is an image enhancer for images captured under dark, shim, or low light environments. Multiple images captured at the same exposure are needed as input and are then combined to remove noise and create a single enhanced output frame.

3.2.1 Initializing SCameraLowLightProcessor

The processor is initiated from `SCameraProcessManager`. The following `SCameraProcessorParameter` Keys need values to initialize the `SCameraLowLightProcessor`:

- **STILL_SIZE:** Still image size
- **STILL_INPUT_FORMAT:** Still image input format

- **STILL_OUTPUT_FORMAT**: Still image output format

```
SCameraProcessorManager processorManager =
mSCamera.getSCameraProcessorManager();
if(!processorManager.isProcessorAvailable
    (SCameraProcessorManager.PROCESSOR_TYPE_LOW_LIGHT)) {
    //This device does not support Low Light Processor.

    return false;
}
mProcessor = processorManager.createProcessor
    (SCameraProcessorManager. PROCESSOR_TYPE_LOW_LIGHT);
SCameraProcessorParameter param = mProcessor.getParameters();

param.set(SCameraLowLightProcessor.STILL_INPUT_FORMAT, ImageFormat.JPEG);
param.set(SCameraLowLightProcessor.STILL_OUTPUT_FORMAT, ImageFormat.JPEG);
param.set(SCameraLowLightProcessor.STILL_SIZE, jpegsize);
mProcessor.setParameters(param);
mProcessor.initialize();
```

The callbacks from the `SCameraLowLightProcessor` comes through the `SCameraLowLightProcessor.EventCallback` interface. The developer needs to implement the following methods of `EventCallback` Interface:

- **`onError(int code)`:** Called when an error occurs.
- **`onProcessCompleted(Image result)`:** Called when the `requestMultiProcess(List)` is done

Error conditions can be handled in onError, and after process completion, we get the final image.

```
mProcessor.setEventCallback(new SCameraLowLightProcessor.EventCallback() {
    @Override
    public void onProcessCompleted(Image result) {
        //Result image handling needs to be done here

    }
    @Override
    public void onError(int code) {
        //Error code handling

    }
}, mBackgroundHandler);
```


3.2.2 Providing Input

Images with same exposure values are captured and then supplied to the `requestMultiImage()` method of **`SCameraLowLightProcessor`**.

```
MAX_COUNT = param.get(SCameraLowLightProcessor.MULTI_INPUT_COUNT_RANGE)
    .getUpper();
mImageList = new ArrayList<Image>();

mJpegReader = ImageReader.newInstance(jpegsize.getWidth(),
    jpegsize.getHeight(), ImageFormat.JPEG, MAX_COUNT + 1);
mJpegReader.setOnImageAvailableListener(new
   ImageReader.OnImageAvailableListener() {
    @Override
    public void onImageAvailable(ImageReader reader) {
        CUR_COUNT++;
        mImageList.add(reader.acquireNextImage());
        if(CUR_COUNT == MAX_COUNT) {
            mProcessor. requestMultiProcess(mImageList);
            for(Image i : mImageList) i.close();
                mImageList.clear();
            }
        }
    }
}, mBackgroundHandler);
```

3.3 SCameraPanoramaProcessor

`SCameraPanoramaProcessor` allows the developer to capture landscape panoramic views by capturing multiple images and assisting the developer with the capture direction and stitching of the final output image. (`SCameraPanoramaProcessor` has been deprecated from version 1.3.1)

Panoramic views are captured as a series of captured images in any direction. It can be any of the four directions available (i.e., down, up, left, right).Once capture starts in a chosen direction, SCameraPanoramaProcessor provides the user with the following messages to assist in capturing:

- **PANORAMA_DIRECTION_DOWN:** Panorama suggested direction
- **PANORAMA_DIRECTION_LEFT**: Panorama suggested direction
- **PANORAMA_DIRECTION_RIGHT :** Panorama suggested direction
- **PANORAMA_DIRECTION_UP :** Panorama suggested direction

SCameraPanoramaProcessor provides easy-to-use APIs to capture panoramic views. The following are the set of methods provided:

- **start():** Starts the panorama processing
- **stop():** Stops the panorama processing and starts the stitching of the final image
- **cancel():** Cancels the ongoing panorama processing
- **getInputSurface():** Returns the Surface which is used as target of Capture Request to provide the preview data to Processor

3.3.1 Initializing SCameraPanoramaProcessor

The processor instance is taken from SCameraProcessorManager. The following SCameraProcessorParameter keys should have values to initialize the SCameraPanoramaProcessor:

- **STREAM_SIZE:** Input stream size. The preview size
- **STILL_OUTPUT_FORMAT:** Still image output format
- **CAMERA_ID:** Camera device ID which is used to provide the preview data to the Processor. The CameraDevice ID is generally "0" for rear camera and "1" for front camera. It is important to set the **CAMERA_ID** before initializing the processor or else it may result to a flipped image or another unwanted result.

```
SCameraProcessorManager processorManager =
    mSCamera.getSCameraProcessorManager();
if(!processorManager.isProcessorAvailable
    (SCameraProcessorManager.PROCESSOR_TYPE_PANORAMA)) {
    //This device does not support Panorama Processor.

    return false;
}
mProcessor = processorManager.createProcessor
    (SCameraProcessorManager.PROCESSOR_TYPE_PANORAMA);

SCameraProcessorParameter param = mProcessor.getParameters();

param.set(SCameraPanoramaProcessor.STILL_OUTPUT_FORMAT, ImageFormat.JPEG);
param.set(SCameraPanoramaProcessor.STREAM_SIZE, previewSize);
param.set(SCameraPanoramaProcessor.CAMERA_ID, Integer.parseInt(mCameraId));

mProcessor.setParameters(param);
mProcessor.initialize();
```


3.3.2 Providing Input

Unlike HDR and Low-Light, **SCameraPanoramaProcessor** requires the preview data continuously to assist with the direction of capture. It also selectively captures images which it finally stitches to the output image.

```
SCaptureRequest.Builder builder =
    mSCameraDevice.createCaptureRequest(SCameraDevice.TEMPLATE_PREVIEW);
builder.addTarget(mSurface);
builder.addTarget(mProcessor.getInputSurface());
mSCameraCaptureSession.setRepeatingRequest(builder.build(),
   this, mBackgroundHandler);
```

3.3.3 Registering Event Callbacks for SCameraPanoramaProcessor

Callbacks for direction assistance and live preview stitching are provided by the SCameraPanoramaProcessor.EventCallback interface. It is important to implement the interface to get the required callback from SDK. SCameraPanoramaProcesor.EventCallback provides the following interface methods:

- **onDirectionChanged(int direction):** Called when the direction hint is given from the framework library
- **onError(int code):** Called to notify errors to the application. Code supplied is error code
- **onLivePreviewDataStitched(Bitmap data):** Called to send the live stitching preview data
- **onMaxFramesCaptured():** Called when all (maximum) frames required for Panorama are captured and the framework will start the final stitching process
- **onMovingTooFast():** Called to notify the application to display help information to move the device slowly
- **onProcessCompleted(Image result):** Called to deliver the stitched final result. Once onProcessCompleted() callback is received, stop() or cancel() should not be called
- **onRectChanged(int x, int y):** Called when the panorama capturing rectangle has changed. The range of the rectangle is [-1000, -1000 – 1000, 1000]
- **onStitchingProgressed(int progress):** Called to notify the progress of stitching the panorama to the application after complete capture. The range of progress notification is 0 - 100


```
mProcessor.setEventCallback(new SCameraPanoramaProcessor.EventCallback() {

    @Override
    public void onError(int code) {
        //Error handling code

    }

    @Override
    public void onRectChanged(int x, int y) {
        //Rect updates

    }

    @Override
    public void onDirectionChanged(int direction) {
        //Direction updates

    }

    @Override
    public void onStitchingProgressed(int progress) {
        //Stitching progress

    }

    @Override
    public void onLivePreviewDataStitched(Bitmap bitmap) {
        //Live Stitched bitmap

    }

    @Override
    public void onMaxFramesCaptured() {
        //Max frames reached.

    }

    @Override
    public void onMovingTooFast() {
        //Moving fast.

    }

    @Override
    public void onProcessCompleted(Image image) {
        //Final output image

   }
}, mBackgroundHandler);
```


3.4 Using SCameraDepthOfFieldProcessor

Depth of Field (DOF) refers to the area in front and behind of the object in the image that is focused, whereas all other areas are blurred or out of focus. DOF is used to create a dramatic effect in images. There are two ways to describe the qualities of depth of field — shallow DOF or deep DOF. Shallow DOF is when the included focus range is very narrow, which ranges from a few inches to several feet. Deep DOF is when the included range is a couple of yards to infinity. Currently, `SCameraDepthOfFieldProcessor` supports only shallow DOF.

The **`SCameraDepthOfFieldProcessor`** can be used to take two images as input. One image is focused to user selection and the other is at infinity focus.

The sample application implements the following features:

- Depth of Field Effect for captured images

3.4.1 Initializing SCameraDepthOfFieldProcessor

To initialize the `SCameraDepthOfFieldProcessor`:

1. Get the `SCameraDepthOfFieldProcessor` instance by calling `SCameraProcessorManager .getProcessorInstance()`.
2. Get the current settings for `SCameraDepthOfFieldProcessor` by calling `getParameters()` and call `setParameters()` to save the settings modified on the `SCameraProcessorParameter`.
3. Call `initialize()` to initialize the processor instance.

```
SCameraProcessorManager processorManager =
    mSCamera.getSCameraProcessorManager();
if(!processorManager.isProcessorAvailable
    (SCameraProcessorManager.PROCESSOR_TYPE_DEPTH_OF_FIELD)) {
    //This device does not support DOF Processor.

    return false;
}
mProcessor = processorManager.createProcessor
    (SCameraProcessorManager.PROCESSOR_TYPE_DEPTH_OF_FIELD);

SCameraProcessorParameter param = mProcessor.getParameters();

param.set(SCameraDepthOfFieldProcessor.STILL_INPUT_FORMAT, ImageFormat.JPEG);
param.set(SCameraDepthOfFieldProcessor.STILL_OUTPUT_FORMAT, ImageFormat.JPEG);
```



```
param.set(SCameraDepthOfFieldProcessor.STILL_SIZE, jpegsize);
mProcessor.setParameters(param);
mProcessor.initialize();
```

3.4.2 Providing Input

To set the input image:

1. Capture two images, one at infinity focus and the other one focused to user selection, and pass them as input to `requestMultiImage ()` as an ImageList.

```
mProcessor.requestMultiImage(mImageList);
```

3.4.3 Registering Event Callbacks for SCameraDepthOfFieldProcessor

To get the output image in the application:

1. Create an `SCameraDepthOfFieldProcessor.EventCallback` callback instance and register it by calling `setEventCallback()` on `SCameraDepthOfFieldProcessor`.
2. Handle the `onProcessCompleted` callback to save the output image.
3. Handle the `onError` callback in case any error has been encountered. The following types of exception messages are defined for `SCameraDepthOfFieldProcessor`:
    - **`NATIVE_PROCESSOR_MSG_OUTFOCUS_ERROR_AF`:** Unable to apply Focus contrast effect. Select an area to focus and try again.
    - **`NATIVE_PROCESSOR_MSG_OUTFOCUS_ERROR_INF`:** Unable to apply Focus contrast effect. Subject too far.
    - **`NATIVE_PROCESSOR_MSG_OUTFOCUS_ERROR_SEGMENTATION`:** Unable to apply Focus contrast effect. Subject not detected.

```
mProcessor.setEventCallback(new SCameraOutFocusProcessor.EventCallback() {
    @Override
    public void onProcessCompleted(Image image) {
          /* Result Image arrived. Save it or do other processing method */
          ....
    }

    @Override
    public void onError(int error) {
          /* Handling Error */
```



```
            ....
        }
}, mBackgroundHandler);
```

3.5 Using SCameraEffectProcessor

The **SCameraEffectProcessor** can be used to apply various filter effects defined by SFilter class on captured images as well as for stream processing to process the camera preview or recording.

3.5.1 Using SCameraFilter Class

The **SCameraFilter** class is used to specify particular filter effects to be used with SCameraEffectProcessor. To use filters, the following permission needs to be declared in AndroidManifest.xml:

```
<uses-permission android:name=
"com.samsung.android.provider.filterprovider.permission.READ_FILTER" />
```

the following permission also needs to be declared in AndroidManifest.xml to use Face AR stickers:

```
<uses-permission android:name=
" com.samsung.android.provider.stickerprovider.permission.READ_STICKER_PROVIDER"
/>
```

To verify if the SCameraFilter feature is supported on the device, please use the **isFeatureEnabled()** method.

isFeatureEnabled(int type) : type value is SCAMERA_FILTER

To get SCameraFilter instance and initialize it:

1. Query available filters by calling SCameraFilterManager().getAvailableFilters().
2. Create an SCameraFilter instance by calling createFilter for SCameraFilterInfo object for desired filter SCameraFilterInfo.

3. Save the filter settings modified from the application by calling `setParameter()` for `SCameraFilter`.

```
List<SCameraFilterInfo> mFilterInfoList =
      mScamera.getSCameraFilterManager().getAvailableFilters();
SCameraFilterInfo filterInfo = mFilterInfoList.get(0);
SCameraFilter mFilter = mSCameraFilterManager.createFilter(filterInfo);
mFilter.setParameter(paramName, progress + paramRange.getLower());
```

After initializing the filter, the **`SCameraFilter`** class can be used for applying image processing techniques according to the needs of an image. It can be used to process Bitmap data using the `processImage()` call.

```
Bitmap outputBitmap = mFilter.processImage(inputBitmap);
```

Input and output file paths can also be passed to `processImage();`

```
mFilter.processImage(inputFile, outputFile);
```

NOTE: If you are working with a SCameraFilter instance with filter type `FILTER_TYPE_FACE_AR` or `FILTER_TYPE_AR EMOJI`, `processImage(...)` will not work. For those filters please use `requestSnapCapture()`.

3.5.2 Initializing SCameraEffectProcessor

To initialize the `SCameraEffectProcessor`:

1. Get the `SCameraEffectProcessor` instance by calling `SCameraProcessorManager .getProcessorInstance().`
2. Get the current settings for `SCameraEffectProcessor` by calling `getParameters()`, and call `setParameters()` to save the settings modified in the `SCameraProcessorParameter.`
3. Call `initialize()` to initialize the processor instance.

```
SCameraProcessorManager processorManager =
    mSCamera.getSCameraProcessorManager();
if(!processorManager.isProcessorAvailable
    (SCameraProcessorManager.PROCESSOR_TYPE_EFFECT)) {
    //This device does not support Effect Processor.

    return false;
}
mProcessor = processorManager.createProcessor
```



```
    (SCameraProcessorManager.PROCESSOR_TYPE_EFFECT);

mFilterInfoList = mSCameraFilterManager.getAvailableFilters();

SCameraProcessorParameter param = mProcessor.getParameters();

param.set(SCameraEffectProcessor.STILL_SIZE, jpegsize);
param.set(SCameraEffectProcessor.STILL_INPUT_FORMAT, ImageFormat.JPEG);
param.set(SCameraEffectProcessor.STILL_OUTPUT_FORMAT, ImageFormat.JPEG);
param.set(SCameraEffectProcessor.STREAM_SIZE, previewSize);
param.set(SCameraEffectProcessor.FILTER_EFFECT,
    mSCameraFilterManager.createFilter(mFilterInfoList.get(1)));
mProcessor.setParameters(param);
mProcessor.initialize();
```

3.5.3 Image Capture with SCameraEffectProcessor

To set the input image, pass the captured image as an input to the `requestProcess()` call. The output image can be saved in registered eventCallback `onProcessCompleted()` method.

```
mProcessor.requestProcess(image);
```

NOTE: If you are working with a filter with `FILTER_TYPE_FACE_AR` filter type, you must use `requestSnapCature()`. Unlike the `requestProcess()`, it does not require input image. Instead, it will provide the snap-shot of a filter effect processed preview image.

```
mProcessor.requestSnapCapture();
```

To get the output image in the application:

1. Create an `SCameraEffectProcessor.EventCallback` callback instance and register it by calling `setEventCallback()` on `SCameraEffectProcessor`.
2. Handle the `onProcessCompleted` callback to save the output image.
3. Handle the `onError` callback in case any error is encountered.

```
mProcessor.setEventCallback(new SCameraEffectProcessor.EventCallback() {
    @Override
    public void onProcessCompleted(Image image) {
        /* Result Image arrived. Save it or do other processing method */
        ....
```



```
    }

    @Override
    public void onError(int i) {
        /* Handling Error */
        ....
    }
}, mBackgroundHandler);
```

3.5.4 Preview Stream processing with SCameraEffectProcessor

In order to do stream processing for camera preview or video recording:

1. Preview surface or media recorder surface is given as output surface for effect processor by calling `setOutputSurface()`.
2. Input surface from `SCameraEffectProcessor` is given as target to `SCaptureRequest`.
3. Processing is started and stopped by calling `startStreamProcessing()` and `stopStreamProcessing()`, `respectively, on SCameraEffectProcessor` on the object.

```
mProcessor.setOutputSurface(mPreviewSurface);
public void startPreview(){
    try {
       // Starts displaying the preview.
       mSCameraSession.setRepeatingRequest(mPreviewBuilder.build(),
           mSessionCaptureCallback, mBackgroundHandler);
       setState(CAMERA_STATE.PREVIEW);
       // must be called after setRepeatingRequest (includes surface of camera
        preview).
        mProcessor.startStreamProcessing();
    } catch (CameraAccessException e) {
        // Handle Error.

    }
}

public void stopPreview(){
    try {
       if (mSCameraSession != null)
          mSCameraSession.stopRepeating();

       if (mProcessor != null && getState() == CAMERA_STATE.PREVIEW)
          mProcessor.stopStreamProcessing();
```



```
        } catch (CameraAccessException e) {
            //Handle Error.
    }
}
```

3.5.5 Video Recording with SCameraEffectProcessor

SCameraEffectProcessor can also be used while video recording from the camera. The developer can choose the effect and apply it directly on video while recording it.

```
private void prepareMediaRecorder() throws IOException {
    mMediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
    mMediaRecorder.setVideoSource(MediaRecorder.VideoSource.SURFACE);
    mMediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);
    mMediaRecorder.setOutputFile(new File(getExternalFilesDir(null),
        "temp.mp4").getAbsolutePath());

    int bitrate = 384000;
    if (mVideoSize.getWidth() * mVideoSize.getHeight() >= 1920 * 1080) {
        bitrate = 14000000;
    } else if (mVideoSize.getWidth() * mVideoSize.getHeight() >= 1280 * 720) {
        bitrate = 9730000;
    } else if (mVideoSize.getWidth() * mVideoSize.getHeight() >= 640 * 480) {
        bitrate = 2500000;
    } else if (mVideoSize.getWidth() * mVideoSize.getHeight() >= 320 * 240) {
        bitrate = 622000;
    }
    mMediaRecorder.setVideoEncodingBitRate(bitrate);

    mMediaRecorder.setVideoFrameRate(MAX_PREVIEW_FPS);
    mMediaRecorder.setVideoSize(mVideoSize.getWidth(), mVideoSize.getHeight());
    mMediaRecorder.setVideoEncoder(MediaRecorder.VideoEncoder.H264);
    mMediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);
    mMediaRecorder.setOrientationHint(getJpegOrientation());
    mMediaRecorder.prepare();
}

private synchronized void recordVideo() {
    setState(CAMERA_STATE.RECORD_VIDEO);

    // sets the surface for recording
    mProcessor.setRecordingSurface(mMediaRecorder.getSurface());
    // Starts recording
    mMediaRecorder.start();
```



```
    mRecordingStartTime = System.currentTimeMillis();
}

private synchronized void stopRecordVideo(boolean isPausing) {

    mProcessor.setRecordingSurface(null);
    // Stops recording
    mMediaRecorder.stop();
    MediaRecorder.reset();

    // Saves recording file

    if (isPausing == false) {
        try {
            prepareMediaRecorder();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    setState(CAMERA_STATE.PREVIEW);
}
```

3.6 Using SCameraHazeRemoveProcessor

The **SCameraHazeRemoveProcessor** can be used to apply haze remove effect on captured images as well as for stream processing to process the camera preview. (SCameraHazeRemoveProcessor has been deprecated from version 1.3.1)

3.6.1 Initializing SCameraHazeRemoveProcessor

To initialize the SCameraHazeRemoveProcessor:

1. Get the SCameraHazeRemoveProcessor instance by calling SCameraProcessorManager.getProcessorInstance().
2. Get the current settings for SCameraHazeRemoveProcessor by calling getParameters(), and call setParameters() to save the settings modified in the SCameraProcessorParameter.
3. Call initialize() to initialize the processor instance.


```
SCameraProcessorManager processorManager =
    mSCamera.getSCameraProcessorManager();
if(!processorManager.isProcessorAvailable
    (SCameraProcessorManager.PROCESSOR_TYPE_HAZE_REMOVE)) {
    //This device does not support Haze remove Processor.

    return false;
}
mProcessor = processorManager.createProcessor
    (SCameraProcessorManager.PROCESSOR_TYPE_HAZE_REMOVE);

mFilterInfoList = mSCameraFilterManager.getAvailableFilters();

param.set(SCameraEffectProcessor.STILL_SIZE, jpegsize);
param.set(SCameraEffectProcessor.STILL_INPUT_FORMAT, ImageFormat.JPEG);
param.set(SCameraEffectProcessor.STILL_OUTPUT_FORMAT, ImageFormat.JPEG);
param.set(SCameraEffectProcessor.STREAM_SIZE, previewSize);
param.set(SCameraEffectProcessor.HAZE_REMOVE_STRENGTH, 3);
mProcessor.setParameters(param);
mProcessor.initialize();
```

3.6.2 Image Capture with SCameraHazeRemoveProcessor

To set the input image, pass the captured image as an input to the `requestProcess()` call. The output image can be saved in registered eventCallback `onProcessCompleted()` method.

```
mProcessor.requestProcess(image);
```

To get the output image in the application:

1. Create an `SCameraHazeRemoveProcessor.EventCallback` callback instance and register it by calling `setEventCallback()` on `SCameraHazeRemoveProcessor`.
2. Handle the `onProcessCompleted` callback to save the output image.
3. Handle the `onError` callback in case any error is encountered.

```
mProcessor.setEventCallback(new SCameraHazeRemoveProcessor.EventCallback() {
    @Override
    public void onProcessCompleted(Image image) {
        /* Result Image arrived. Save it or do other processing method */
        ....
    }
```



```
    @Override
    public void onError(int code) {
        /* Handling Error */
        ....
    }
}, mBackgroundHandler);
```

3.6.3 Preview Stream processing with SCameraHazeRemoveProcessor

To do stream processing for camera preview or video recording:

1. Obtain preview surface from the SCameraHazeRemoveProcessor by calling setOutputSurface().
2. Input the surface from `SCameraHazeRemoveProcessor` to `SCaptureRequest as target`.
3. Start or stop processing by calling `startStreamProcessing()` or `stopStreamProcessing()` of the `SCameraHazeRemoveProcessor` object.

```
mProcessor.setOutputSurface(mPreviewSurface);
public void startPreview(){
    try {
       // Starts displaying the preview.
       mSCameraSession.setRepeatingRequest(mPreviewBuilder.build(),
           mSessionCaptureCallback, mBackgroundHandler);
       setState(CAMERA_STATE.PREVIEW);
       mProcessor.startStreamProcessing();
    } catch (CameraAccessException e) {
        // Handle Error.

    }
}

public void stopPreview(){
    try {
       if (mSCameraSession != null)
          mSCameraSession.stopRepeating();
       if (mProcessor != null && getState() == CAMERA_STATE.PREVIEW)
          mProcessor.stopStreamProcessing();
       } catch (CameraAccessException e) {
          //Handle Error.
    }
}
```


3.7 Using SCameraGifProcessor

SCameraGIFProcessor creates animated GIF from a set of multiple images captured by user. This class inherits from SCameraProcessor. (SCameraGIFProcessor has been deprecated from version 1.3.1)

3.7.1 Initializing SCameraGifProcessor

Processor is initiated from ProcessorManager. Values for the following SCameraProcessorParameter keys are needed to initialize the SCameraGifProcessor:

- **STILL_SIZE** : Size of the GIF output.
- **GIF_FRAME_DURATION** : Desired frame duration of the GIF output.

```
SCameraProcessorManager processorManager =
mSCamera.getSCameraProcessorManager();
if(!processorManager.isProcessorAvailable
    (SCameraProcessorManager.PROCESSOR_TYPE_GIF)) {
    //This device does not support GIF Processor.
    return false;
}
mProcessor = processorManager.createProcessor
    (SCameraProcessorManager.PROCESSOR_TYPE_GIF);
SCameraProcessorParameter param = mProcessor.getParameters();

param.set(SCameraGifProcessor.STILL_SIZE, new Size(GIF_WIDTH, GIF_HEIGHT);
param.set(SCameraGifProcessor.GIF_FRAME_DURATION, 100); //100ms

mProcessor.setParameters(param);
mProcessor.initialize();
```

3.7.2 Providing Input

Multiple images are captured and then supplied as a list of bitmaps to the requestMultiProcess() method of SCameraGifProcessor.

- **requestMultiProcess(List<Bitmap> data):** Creates an animated GIF. After calling this method, bitmap data will be added as a GIF image frame.

```
mJpegReader = ImageReader.newInstance(jpegsize.getWidth(), jpegsize.getHeight(),
     ImageFormat.JPEG, MAX_COUNT + 1);
mJpegReader.setOnImageAvailableListener(new
```



```
     ImageReader.OnImageAvailableListener() {
     @Override
     public void onImageAvailable(ImageReader reader) {
        mImageList.add(decodeToBitmap(reader.acquireNextImage();

        if(mImageList.size() == MAX_COUNT)
            mProcessor.requestMultiProcess(mImageList);
    }
}, mBackgroundHandler);
```

3.7.3 Registering Event Callback for SCameraGifProcessor

The callbacks from the `SCameraGifProcessor` is provided by `SCameraGifProcessor.EventCallback` interface. It is required that the following methods of the `EventCallback` interface should be implemented:

- **`onError(int code)`:** Called when error occurred.
- **`onProcessCompleted(byte[] data)`:** Called as **`requestMultiProcess(List<Bitmap>)`** is finished.

Error conditions can be handled in `onError` and after process completion final GIF image outputted as byte array.

```
mProcessor.setEventCallback(new SCameraGifProcessor.EventCallback() {
    @Override
    public void onProcessCompleted(byte[] data) {
        //Result Image arrived. Save it
    }

    @Override
    public void onError(int code) {
        //Error handling code
    }
}, mBackgroundHandler);
```

4. Using Image

Image is a framework which works in heterogeneous environments in which application developers will get the maximum benefit in terms of performance. This framework supports

various image processing and mathematical operations. (`Image` package has been deprecated from version 1.3.1)

NOTE: The type for **isFeatureEnabled(int type)** is SCAMERA_IMAGE

Before calling any of the image package methods, an `SCamera` instance has to be created and initialized in order to load native side dependent libraries.

```
SCamera mScamera = new SCamera();
try {
    mScamera.initialize(this);
} catch (SsdkUnsupportedException e) {
    e.printStackTrace();
}
```

4.1 Using SCameraImage

SCameraImage allows creating an image buffer for processing using Image package APIs from image byte buffer or from a stored file. It can support high quality image for processing.

Advantage of SCameraImage over Android Bitmap:

- `SCameraImage` supports very large image buffers whereas Bitmap doesn't support higher image resolutions.

An instance can be created from an `SCameraImage` with a default input format or with the following supported input formats:

- `FORMAT_RGB24`
- `FORMAT_NV21`
- `FORMAT_YUYV`

The default format is FORMAT_NV21. The preferred format is FORMAT_NV21 for high algorithm performance.

The **SCameraImage** class provides methods to get and set the image color and image pixels.

It provides APIs for getting the processed image bitmap and save them in jpeg or raw format.

- **`Bitmap getBitmap()`** – Returns the processed bitmap
- **`saveAsJpeg(String pathname, int quality)`** - Saves the SImage at the specified path with the given quality
- **`saveAsRaw(String pathname)`** - Saves raw format SImage at the specified path

The following code demonstrates the usage of the SCameraImage class:

```
public class MainActivity extends Activity {

    SCamera mScamera;
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mScamera = new SCamera();
        try {
            mScamera.initialize(this);
        } catch (SsdkUnsupportedException e) {
        }
        SCameraImage image;

        image=new SCameraImage(mImagePath,SImage.FORMAT_DEFAULT);
        //perform processing using Image package API

        . . .
        image.saveAsJpeg("/sdcard/DCIM/sample.jpg" 95);
        image.saveAsRaw("/sdcard/DCIM/sample.raw");
    }
}
```

4.2 Using SCameraImageCore

The SCameraImageCore class provides support for image processing. It provides support for sobel, median filter, warp, and histogram equalization. It also provides methods to convert the images to different formats. SImage has to be created to whatever image processing algorithm is applied. The default format is FORMAT_NV21.

SCameraImage's buffer will be released when the instance is finalized; however, calling **release()** method is recommended for efficient memory management.

The supported image package APIs are:

4.2.1 Sobel

The Sobel filter is an image processing technique to create an image that emphasizes edges and transitions. It calculates the gradient of the source image by applying the Sobel operator in requested direction. An image gradient is a directional change in the intensity or color of an image.

To apply Sobel to the image:

1. Perform Sobel operation by calling `SCameraImageCore: processSobel(SCameraImage image, int threshold)`.
2. Get the output bitmap by calling `SCameraImage:getBitmap()` and display or save it in the file system.
3. Release `SCameraImage` instances.

```
mOutputImage = SCameraImageCore.processSobel(mInputImage,100);
mOutputBitmap = getOutputBitmap(mOutputImage);

if(mOutputBitmap == null) {
    Toast.makeText(getApplication().getBaseContext(), "Failed to apply effect",
        Toast.LENGTH_SHORT).show();
    mOutputImage.release();
    return false;
} else {
    mOutputView.setImageBitmap(mOutputBitmap);
}
```

NOTE: For more information, refer to the SampleCamera app.

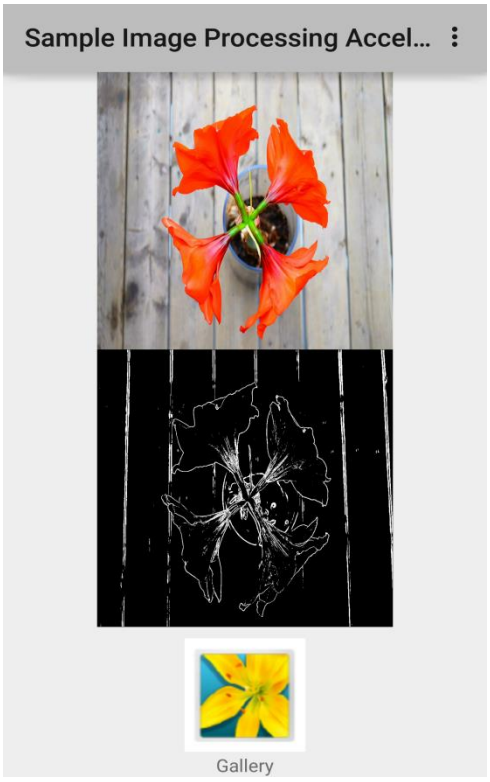

Figure 5: Sobel

4.2.2 Median

The median filter is an image processing technique which is often used to remove noise. Each output pixel contains the median value in the NxN neighborhood around the corresponding pixel in the input image. N would be the kernel size, in this case, which should always be an odd number. The median is calculated by first sorting all the pixel values from the surrounding neighborhood into numerical order and then replacing the pixel being considered with the middle pixel value. Hence, in a very large kernel size, processing will be slower.

To apply Median to the image:

1. Perform Median operation by calling `SCameraImageCore: processMedian(SCameraImage image, int size)`.
2. Get the output bitmap by calling `SCameraImage:getBitmap()` and display or save it in the file system.
3. Release `SCameraImage` instances.

```
mOutputImage = SCameraImageCore.processMedian(mInputImage,5);
mOutputBitmap = getOutputBitmap(mOutputImage);

if(mOutputBitmap == null) {
    Toast.makeText(getApplication().getBaseContext(), "Failed to apply effect" ,
        Toast.LENGTH_SHORT).show();
    mOutputImage.release();
    return false;
} else {
    mOutputView.setImageBitmap(mOutputBitmap);
}
```

NOTE: For more information, refer to the SampleCamera app.

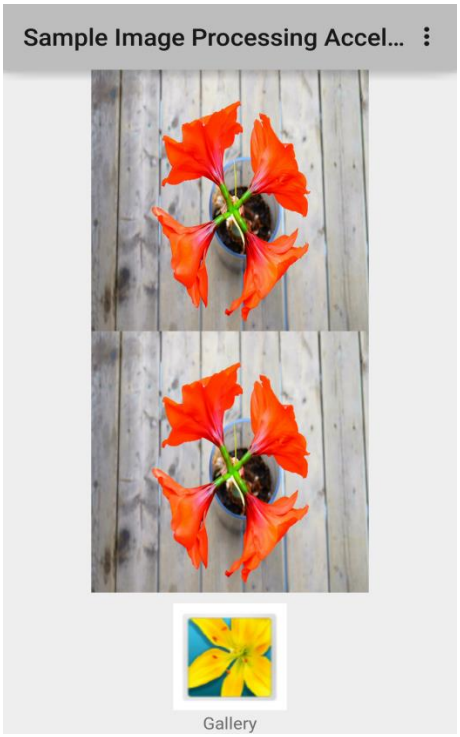

Figure 6: Median

4.2.3 Contrast Enhancement

Enhance the contrast of the given source image and generate an output image. The API `enhanceContrast` takes two input parameters, contrast factor, which denotes the percentage of contrast by which the image needs to be enhanced (0 to 1), and pivot, which represents the color intensity value about which the contrast needs to be enhanced. It should range from 0 to 255.

To apply Contrast to the image:

1. Perform Contrast operation by calling `SCameraImageCore: enhanceContrast( SCameraImage image, int pivot, float contrast).`
2. Get the output bitmap by calling `SCameraImage:getBitmap()` and display or save it in the file system.
3. Release `SCameraImage` instances.

```
mOutputImage = SCameraImageCore.enhanceContrast(mInputImage,100, 0.5f);
mOutputBitmap = getOutputBitmap(mOutputImage);
```



```
if(mOutputBitmap == null) {
    Toast.makeText(getApplication().getBaseContext(), "Failed to apply effect" ,
       Toast.LENGTH_SHORT).show();
    mOutputImage.release();
    return false;
} else {
    mOutputView.setImageBitmap(mOutputBitmap);
}
```

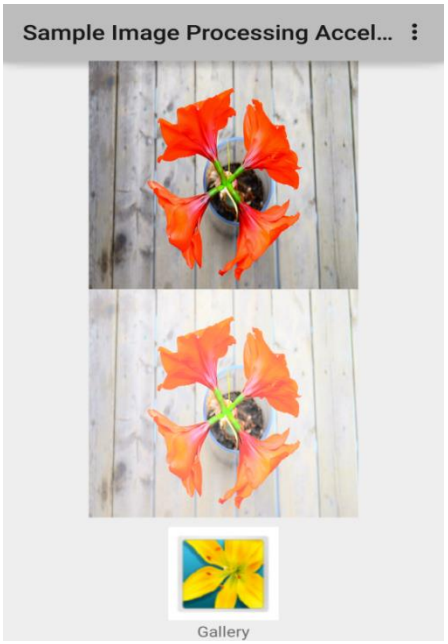

Figure 7: Contrast

4.2.4 Equalize Histogram

It is an image intensifying adjustment technique through which gain in contrast can be seen in output image.

To apply Histogram to the image:

1. Perform Histogram operation by calling `SCameraImageCore: equalizeHistogram( SCameraImage image).`
2. Get the output bitmap by calling `SCameraImage:getBitmap()` and display or save it in the file system.
3. Release `SCameraImage` instances.


```
mOutputImage = SCameraImageCore.equalizeHistogram(mInputImage);
mOutputBitmap = getOutputBitmap(mOutputImage);

if(mOutputBitmap == null) {
    Toast.makeText(getApplication().getBaseContext(), "Failed to apply effect" ,
        Toast.LENGTH_SHORT).show();
    mOutputImage.release();
    return false;
} else {
    mOutputView.setImageBitmap(mOutputBitmap);
}
```

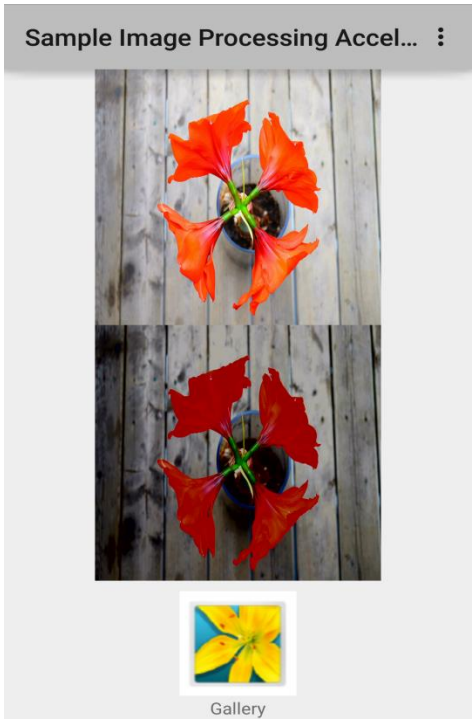

Figure 8: Equalize Histogram

4.2.5 Warp Affine

Warp Affine applies an affine transformation to an image.

To apply Warp to the image:

1. Create an SCameraImageMatrix instance and set the required transformation values to the matrix. Pass the matrix and input image to `warpAffine()`. The APIs

`calculateAffineRotation`, `calculateAffineTranslation`, `calculateAffineScale`, and `calculateAffineSkew` can be used to set the matrix values required for transformation.

2. Get the output bitmap by calling `SCameraImage:getBitmap()` and display or save it in RAW/JPEG format in the file system.
3. Release `SCameraImage` instances.

```
SCameraImageMatrix mat = SCameraImageCore.getDefaultAffineMatrix();
SCameraImageCore.calculateAffineScale(mat,0.5f,0.5f);
SCameraImageCore.calculateAffineSkew(mat,0.9f,0.8f);
SCameraImageCore.calculateAffineRotation(mat,30,400,600);

mOutputImage = SCameraImageCore.warpAffine(mInputImage,mat);
mOutputBitmap = getOutputBitmap(mOutputImage);
mat.release();

if(mOutputBitmap == null) {
    Toast.makeText(getApplication().getBaseContext(), "Failed to apply effect" ,
        Toast.LENGTH_SHORT).show();
    mOutputImage.release();
    return false;
} else {
    mOutputView.setImageBitmap(mOutputBitmap);
}
```


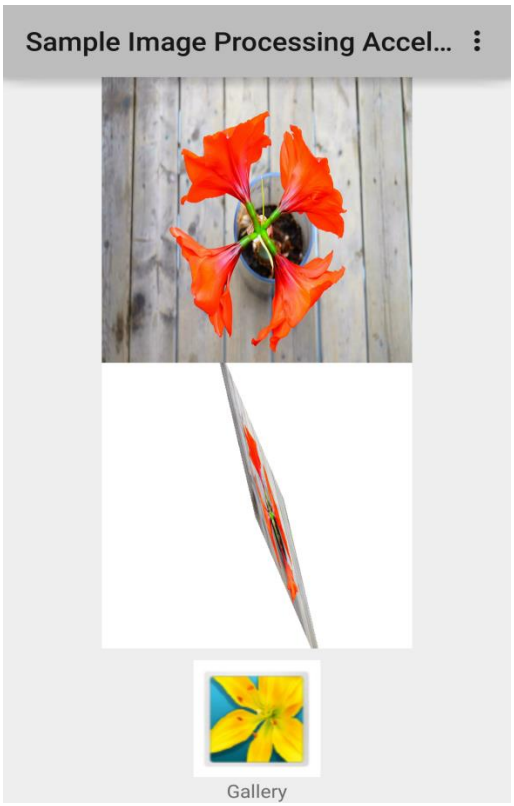

Figure 9: Warp Affine

4.2.6 Spatial Filter

Spatial filtering is a neighborhood operation where the value of a pixel in the output image is a weighted sum of the corresponding neighborhood pixels of input. The weights are defined in the mask of the filter. The mask could be laplacian, hiboost, Gaussian, etc.

To apply Spatial filter to the image:

1. Perform Spatial operation by calling `SCameraImageCore: filterSpatial(SCameraImage image, SCameraImageMatrix matrix).`
2. Get the output bitmap by calling `SCameraImage:getBitmap()` and display or save it in the file system.
3. Release `SCameraImage` instances.


```
int N =7;
SCameraImageMatrix matrix = new SCameraImageMatrix(N, N);
for(int i =0;i<N;i++) {
    for(int j =0;j<N;j++) {
        matrix.setAt(i,j,1);
    }
}

mOutputImage = SCameraImageCore.filterSpatial(mInputImage,matrix);
mOutputBitmap = getOutputBitmap(mOutputImage);

matrix.release();
if(mOutputBitmap == null) {
    Toast.makeText(getApplication().getBaseContext(), "Failed to apply effect" ,
       Toast.LENGTH_SHORT).show();
    mOutputImage.release();
    return false;
} else {
    mOutputView.setImageBitmap(mOutputBitmap);
}
```

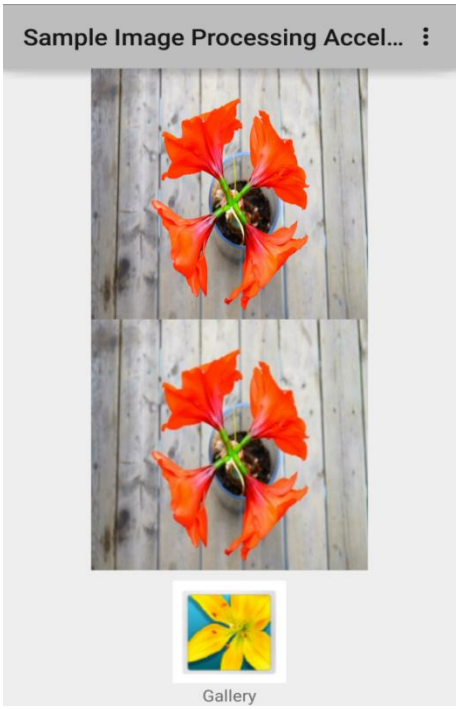

Figure 10: Spatial Filter

4.2.7 Color conversion

Supports color conversion from one format to another. The following color conversions are supported:

- RGB24 to YUYV
- YUYV to RGB24
- NV21 to YUYV
- YUYV to NV21
- RGB24 to NV21
- NV21 to RGB24

To apply Color conversion to the image:

- Create instance of `SCameraImage`. i.e., `new SCameraImage (String filepath, int format)`
- Perform Color conversion operation by calling `SCameraImageCore: convertImageFormat(SCameraImage image, int format)`
- Get the output bitmap by calling `SCameraImage:getBitmap()` and display on the output view or save the file in raw format or in jpeg.
- Release `SCameraImage` instances.

```
SCameraImage inImage =new SCameraImage("/sdcard/DCIM/test.jpg",
                                                SCameraImage.FORMAT_NV21);
SCameraImage outImage = SCameraImageCore.convertImageFormat(inImage,
SCameraImage. FORMAT_RGB24);

mOutputView.setImageBitmap(outImage.getBitmap());

inImage.release();
outImage.release();
```

4.3 Using SCameraImageMatrix

SCameraImageMatrix is a template class for Matrix of floating point values. All operations on Matrix are supported. Values can be accessed by using getAt() and setAt(). The array of Matrix value can be used to construct a matrix by using setValues() .

The matrix value can be:

Hiboost $\begin{Bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{Bmatrix}$

Average

$$\begin{Bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{Bmatrix}$$

Laplacian

$$\begin{Bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{Bmatrix}$$

Values of matrix can vary based on the operations or intensity.

```
int N =7;
SCameraImageMatrix matrix = new SCameraImageMatrix (N, N);

for(int i =0;i<N;i++) {
    for(int j =0;j<N;j++) {
        matrix.setAt(i,j,0.11f);
    }
}

SImage spatial = SCameraImageCore.filterSpatial(image,matrix);
```


Copyright

Though every care has been taken to ensure the accuracy of this document, Samsung Electronics Co., Ltd. cannot accept responsibility for any errors or omissions or for any loss occurred to any person, whether legal or natural, from acting, or refraining from action, as a result of the information contained herein. Information in this document is subject to change at any time without obligation to notify any person of such changes.

Samsung Electronics Co. Ltd. may have patents or patent pending applications, trademarks copyrights or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give the recipient or reader any license to these patents, trademarks copyrights or other intellectual property rights.

No part of this document may be communicated, distributed, reproduced or transmitted in any form or by any means, electronic or mechanical or otherwise, for any purpose, without the prior written permission of Samsung Electronics Co. Ltd.

The document is subject to revision without further notice.

All brand names and product names mentioned in this document are trademarks or registered trademarks of their respective owners.

For more information, please visit http://developer.samsung.com/