NSP32 SDK®

Android Developer Guide

ver 1.7

nanoLambda

IMPORTANT NOTICE

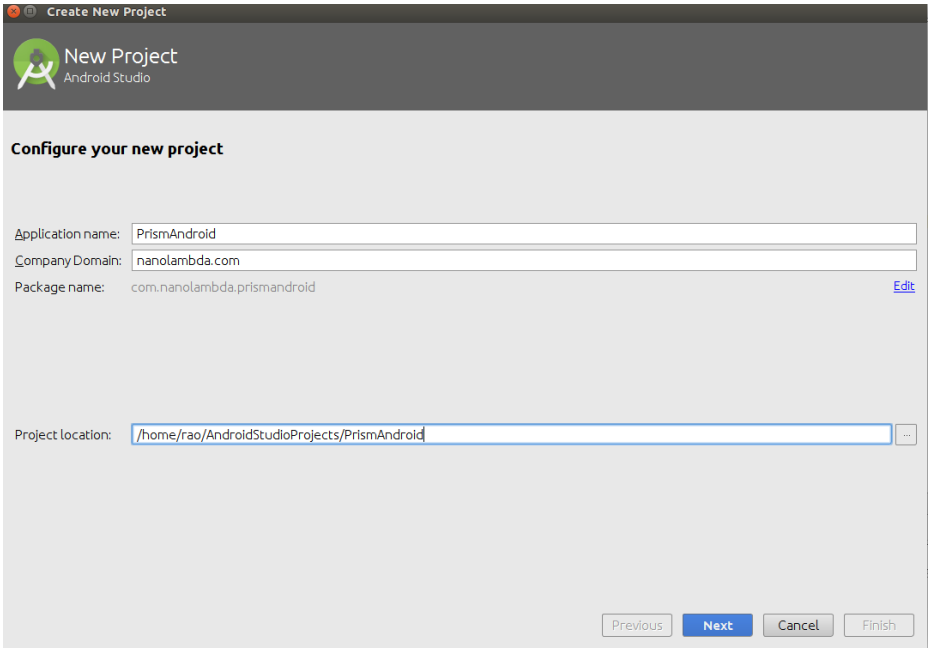
nanoLambda Korea and its affiliates (“nanoLambda”) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to nanoLambda’s terms and conditions of sale supplied at the time of order acknowledgment. Customers are responsible for their products and applications using any nanoLambda products. nanoLambda does not warrant or represent that any license, either express or implied, is granted under any nanoLambda patent right, copyright, mask work right, or other nanoLambda intellectual property right relating to any combination, machine, or process in which nanoLambda products or services are used. Information published by nanoLambda regarding third-party products or services does not constitute a license from nanoLambda to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from nanoLambda under the patents or other intellectual property of nanoLambda. Reproduction of nanoLambda information in nanoLambda documents or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. nanoLambda is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions. Resale of nanoLambda products is not allowed without written agreement. Decompiling, disassembling, reverse engineering or attempt to reconstruct, identify or discover any source code, underlying ideas, techniques or algorithms are not allowed by any means. nanoLambda products are not authorized for use in safety-critical applications. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of nanoLambda products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by nanoLambda. Further, buyers must fully indemnify nanoLambda and its representatives against any damages arising out of the use of nanoLambda products in such safety-critical applications. This Notice shall be governed by and construed in accordance with the laws of Korea, without reference to principles of conflict of laws or choice of laws. All controversies and disputes arising out of or relating to this Notice shall be submitted to the exclusive jurisdiction of the Daejeon District Court in Korea as the court of first instance.

Application Programming Interface (API) For NSP32 Spectral Sensor

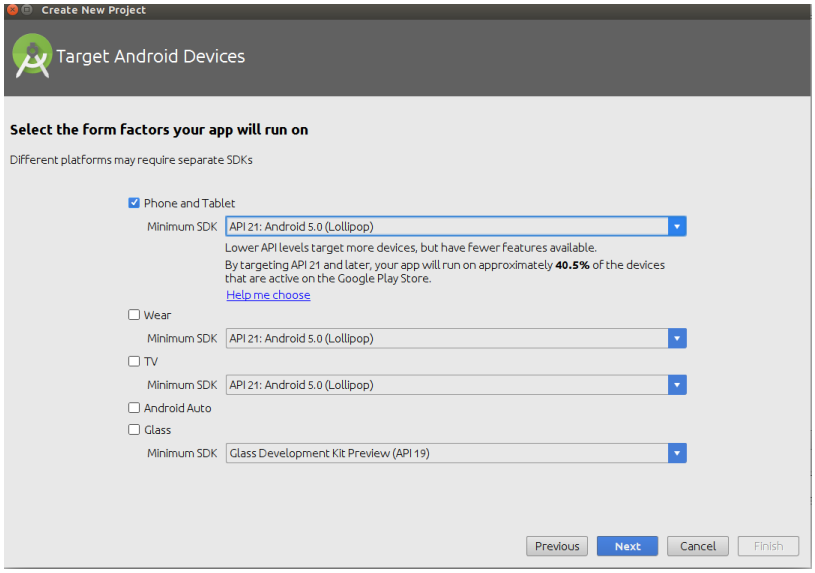
NSP32 application programming interface (API) is a list of all classes that are part of the NSP32 Software Development Kit (SDK). It includes all libraries, classes, and interfaces, along with their methods, fields, and constructors. These prewritten classes provide a tremendous amount of functionality to a programmer. A programmer should be aware of these classes and should know how to use them. If you browse through the list of packages in the API, you will observe that there are packages written for reading data from NSP32 spectral sensor, connecting single or multiple sensors, managing input and output, getting spectrum data for single or multiple sensors, and many more. Please browse this manual for complete list of available functions and their descriptions to see how they can be used.

Developing Android Example:

Insert Application, company Domain, and Project Location. After that click next. You can choose your own application, company domain, and projection location.

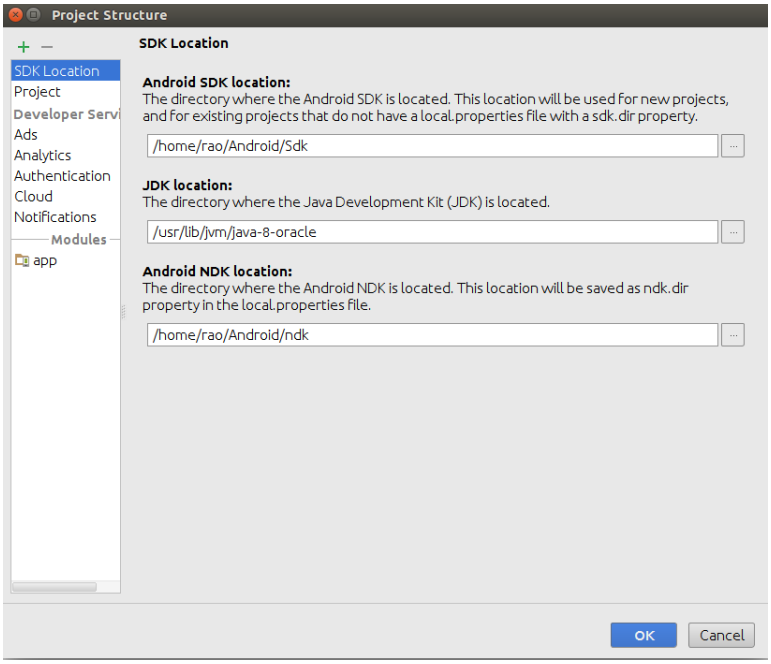


We will work with Phone and Tablet so select this option and select Minimum SDK from drop-down menu. (API-21: Android 5.0 (Lollipop)). After that Click Next.

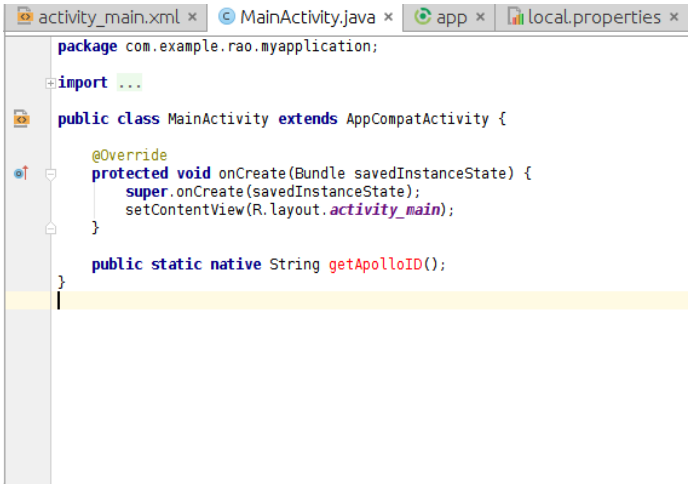


Select Empty Activity and click next and then click finish.

1. **NDK Path:** First of all write the NDK path in local.properties or you can set it by File->Project Structure->SDK Location. In the last option set the NDK location.



2. Write one native function in MainActivity. This function will be interface with native libraries. Let's take an example of getting sensorID from apollo. This function can have any name, it doesn't need to be name from Crystal-libraries.

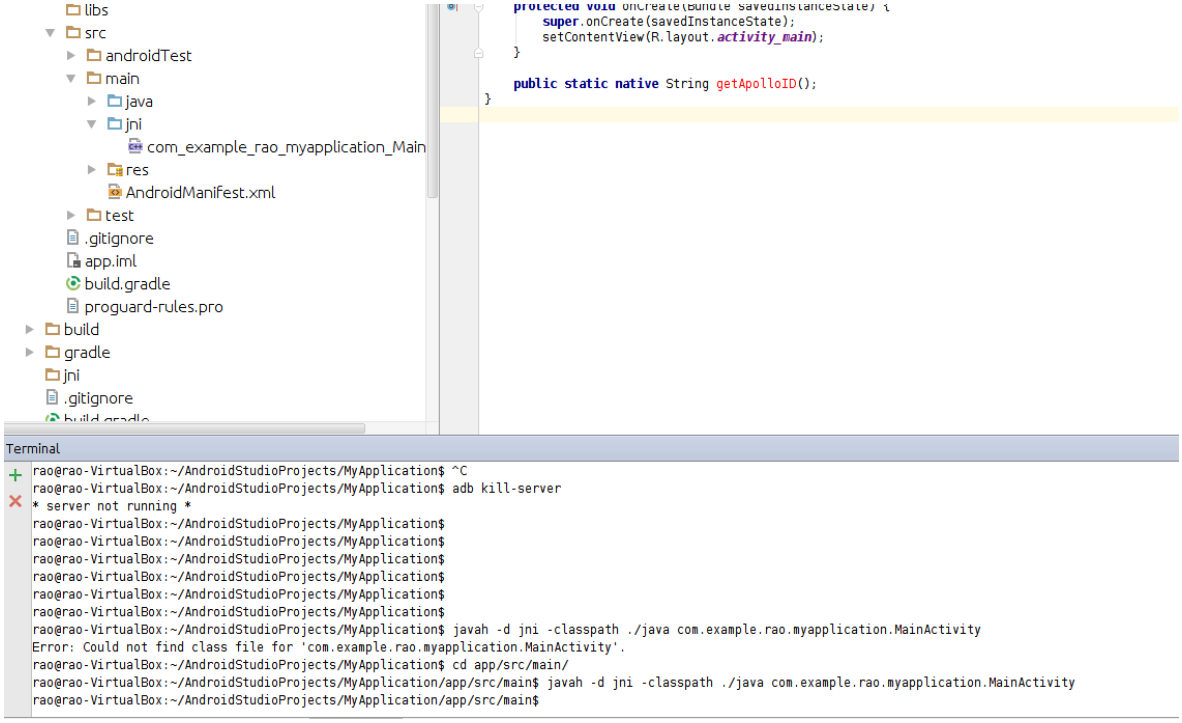


3. Now open the terminal in the Android studio and write this command.

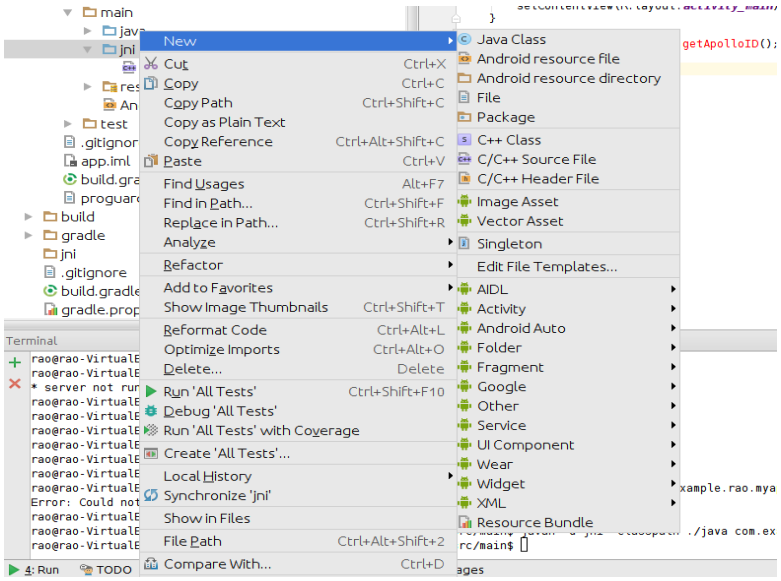
   cd app/src/main

   javah -d jni -classpath ./java com.example.nanoandroid.MainActivity

The above command will make one directory with the name of "**jni**" in app->src->main and in that directory you will see one header file with the name of your package.

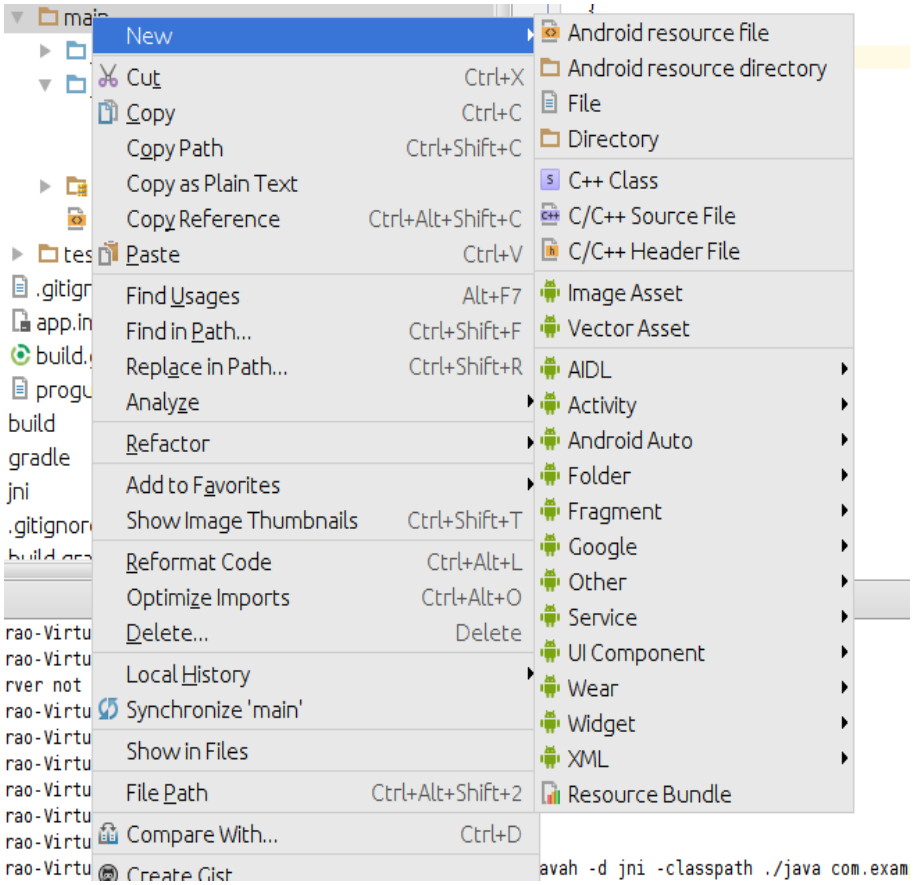


4. In jni directory make one main.cpp file. Right click on the jni and select New->C/C++ source file.



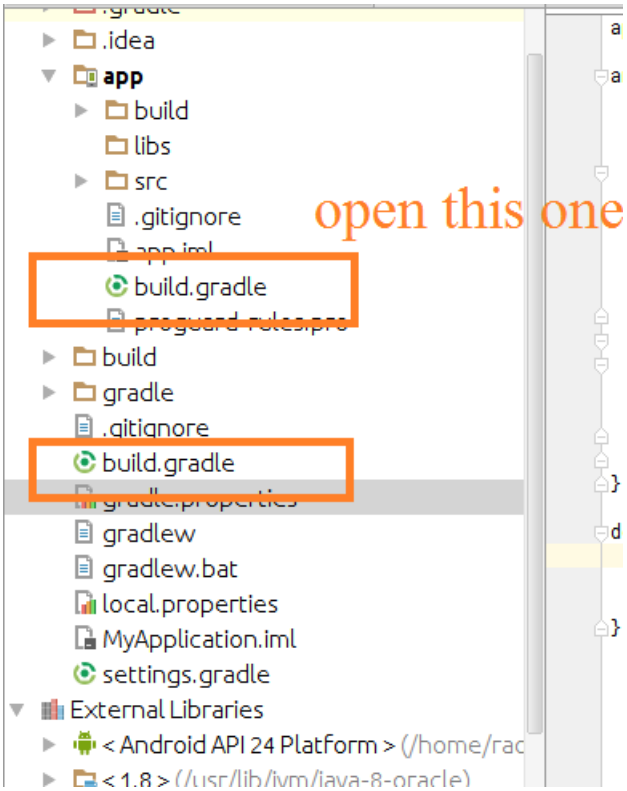
5. In the main.cpp file. Include the header file which we generated in step 9.

6. In the function write your C/C++ code.
7. Make one new folder in Project/app/src/main with the name of **jniLibs**. Right click on the main and select New->Directory.



8. In jniLibs Folder, create one more folder with the specific architecture. For our case it is "**armeabi-v7a**". In this folder copy all the cross-compiled libraries.

9. Now open build.gradle file. Notice that there are two build.gradle in this project. Remember to open the one under app folder.



10. In this build.gradle file, we need to tell where our native libraries and header files are. For user convenience, include and libraries of the compiled tool-chain is also included in the jni and jniLibs folders.

```
def libsDir = projectDir.path + "/src/main/jniLibs/"
productFlavors {
    arm {
        ndk {
            moduleName "MyAppNative"
            abiFilters "armeabi-v7a"
            ldLibs "log"
            ldLibs "android"
            stl "gnustl_shared"
            cFlags "-fexceptions"
            ldLibs libsDir + "armeabi-v7a/libusbnok.a"
            ldLibs libsDir + "armeabi-v7a/libcrystalbase_android.a"
            ldLibs libsDir + "armeabi-v7a/libcrystalcore_android.a"
            ldLibs libsDir + "armeabi-v7a/libcrystalport_android.a"
            ldLibs libsDir + "armeabi-v7a/libgsl.a"
            ldLibs libsDir + "armeabi-v7a/libgslcblas.a"
        }
    }
}
```

```
buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
}
sourceSets.main {
    jni.srcDir 'src/main/jni'
    jni.srcDir 'src/main/jni/device'
    jni.srcDir 'src/main/jni/base'
    jni.srcDir 'src/main/jni/efm32com'
    jni.srcDir 'src/main/jni/'
    jni.srcDir 'src/main/jni/gsl'
    jni.srcDir 'src/main/jni/toolchain_include/c++/4.9'
    jniLibs.srcDir '/src/main/jniLibs/armeabi-v7a/lib'
    assets.srcDirs = ['src/main/assets']
}
```

17. Open the MainActivity and there you will see the function which we define earlier.

```
package com.example.rao.myapplication;

import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        this.getApolloID();
    }
```

18. Android works with shared libraries (.so). In the .apk, you will see .so files. in build.gradle we named our module **"MyAppNative"**

```
def libsDir = projectDir.path + "/src/main/jniLibs/"
productFlavors {
    arm {
        ndk {
            moduleName "MyAppNative"
            abiFilters "armeabi-v7a"
            ldLibs "log"
            ldLibs "android"
            stl "gnustl_shared"
            cFlags "-fexceptions"
            ldLibs libsDir + "armeabi-v7a/libusbnok.a"
            ldLibs libsDir + "armeabi-v7a/libcrystalbase_android.a"
            ldLibs libsDir + "armeabi-v7a/libcrystalcore_android.a"
            ldLibs libsDir + "armeabi-v7a/libcrystalport_android.a"
            ldLibs libsDir + "armeabi-v7a/libgsl.a"
            ldLibs libsDir + "armeabi-v7a/libgslcblas.a"
        }
    }
}
```

In MainActivity file, we need to load this file.

```
public class MainActivity extends AppCompatActivity {

    static{
        System.loadLibrary("MyAppNative");
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        this.getApolloID();
    }

    public static native String getApolloID();
}
```

Note: In case it's give error undefined reference to std::ios_base then use this changing in build.gradle

```
def libsDir = projectDir.path + "/src/main/jniLibs/"
productFlavors {
    arm {
        ndk {
            moduleName "MyAppNative"
            abiFilters "armeabi-v7a"
            ldLibs "log"
            ldLibs "android"
            stl "gnustl_shared"
            cFlags "-fexceptions"
            ldLibs libsDir + "armeabi-v7a/libusbnok.a"
            ldLibs libsDir + "armeabi-v7a/libcrystalbase_android.a"
            ldLibs libsDir + "armeabi-v7a/libcrystalcore_android.a"
            ldLibs libsDir + "armeabi-v7a/libcrystalport_android.a"
            ldLibs libsDir + "armeabi-v7a/libgsl.a"
            ldLibs libsDir + "armeabi-v7a/libgslcblas.a"
            ldLibs "path/to/ndk/sources/cxx-stl/gnu-libstdc++/4.9/libs/armeabi-v7a/libgnustl_static.a"
        }
    }
}
```

```
buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
}
sourceSets.main {
    jni.srcDir 'src/main/jni'
    jni.srcDir 'src/main/jni/device'
    jni.srcDir 'src/main/jni/base'
    jni.srcDir 'src/main/jni/efm32com'
    jni.srcDir 'src/main/jni/'
    jni.srcDir 'src/main/jni/gsl'
    jni.srcDir 'src/main/jni/toolchain_include/c++/4.9'
    jniLibs.srcDir '/src/main/jniLibs/armeabi-v7a/lib'
    assets.srcDirs = ['src/main/assets']
    jni.srcDir "path/to/ndk/sources/cxx-stl/gnu-libstdc++/4.9/libs/armeabi-v7a/include"
}
```

Calibration files:

Assets provide a way to include arbitrary files like text, xml, fonts, music, and video in your application. If you try to include these files as "resources", Android will process them into its resource system and you will not be able to get the raw data. If you want to access data untouched, Assets are one way to do it.

Assets added to your project will show up just like a file system that can read from by your application using AssetManager.

Android has a dedicated folder in its project organization that stores all these files.

So user need to copy sensor calibration files in asset folder and example will load it in the runtime.

```
if (device != null) {
    try {
            String[] f = null;
            try {
                f = getAssets().list("");
            } catch (IOException e) {
                e.printStackTrace();
            }
            for (String f1 : f) {


                if (f1.equals("images") == true || f1.equals("webkit") == true || f1.equals("sounds")) {
                    Log.v("names", f1);
                } else {
                    String localPath_dark = this.getFilesDir().getAbsolutePath() + "/" + f1;
                    ApolloTools.copyFilterResponseFile(f1, localPath_dark, this.getBaseContext());
                }
            }
            GettingBackgroundFilterData("Simple");

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```