E3 Scanner SDK for Android

Version	Date	Author	Modify
V1.0	2020/07/08	xchengtech	First veriosn
V1.1	2020/07/31	xchengtech	Add scanner intent
V1.2	2020/09/17	xchengtech	Support code scanning switch control
V1.3	2020/12/03	xchengtech	Update library interface

Purpose

The purpose of this document release, that guid appliaction development with e3 scanner. This scanner SDK of android provide developers, with a comprehensive set of tools to develop 1D and 2D code scanning applications

Directory

- Native so
- The associated java class of the local library
- Interface implement sample
- Scanner intent

Native so

Description:

All interface for scanner and decode.

Name:

libXCScanner.so

The associated java class of the local library

Description:

The associated java class of the local library(libXCScanner.so).

Associated java class code，and all interface explanation(XCScanner.java) :

```
package com.xcheng.scannere3;

public class XCScanner {
    static
    {
        // Load JNI share libraray(libXCScanner.so)
        System.loadLibrary("XCScanner");
    }

    // Native scanner work mode definition(support work mode)
    public static final int WORKMODE_SINGLESHOT = 0;
    public static final int WORKMODE_CONTINUOUS_SIMPLE = 1;
    public static final int WORKMODE_CONTINUOUS_DISTINCT = 2;
```

```
// Native code type definition(support code type)
public static final int E3_SYM_DOTCODE      = 0;
public static final int E3_SYM_EAN13        = 1;
public static final int E3_SYM_EAN8         = 2;
public static final int E3_SYM_UPCA         = 3;
public static final int E3_SYM_UPCE         = 4;
public static final int E3_SYM_CODE11       = 5;
public static final int E3_SYM_CODE32       = 6;
public static final int E3_SYM_CODE39       = 7;
public static final int E3_SYM_CODE93       = 8;
public static final int E3_SYM_CODE128      = 9;
public static final int E3_SYM_PDF417       = 10;
public static final int E3_SYM_MICROPDF417  = 11;
public static final int E3_SYM_QRCODE       = 12;
public static final int E3_SYM_DATAMATRIX   = 13;
public static final int E3_SYM_CODABAR      = 14;
public static final int E3_SYM_AZTEC        = 15;
public static final int E3_SYM_GS1_128      = 16;
public static final int E3_SYM_GS1_DATABAR  = 17;
public static final int E3_SYM_GS1_LIMITED  = 18;
public static final int E3_SYM_GS1_EXPANDED = 19;
public static final int E3_SYM_TRIOPTIC     = 20;
public static final int E3_SYM_ITF25        = 21;
public static final int E3_SYM_MATRIX25     = 22;
public static final int E3_SYM_IATA25       = 23;
public static final int E3_SYM_INDUSTRIAL25 = 24;
public static final int E3_SYM_CODABLOCK_F  = 25;
public static final int E3_SYM_CODABLOCK_A  = 26;
public static final int E3_SYM_MSI          = 27;
public static final int E3_SYM_COMPOSITE    = 28;
public static final int E3_SYM_TELEPEN      = 29;
public static final int E3_SYM_MAXICODE     = 30;
public static final int E3_SYM_HANXIN       = 31;
public static final int E3_SYM_USPS_4_STATE = 32;
public static final int E3_SYM_HK25         = 33;
public static final int E3_SYM_GRID_MATRIX  = 34;

// Java callbacks invoked by
JNI public interface Result {
    // Called to play scan
    beep void scanBeep();

    // Called to start new scan
    round void scanStart();

    // Called to report scan result
    void scanResult(String sym,String content);
}

private XCScanner()
{
}

// Create new scanner instance
public native static XCScanner newInstance();

// Delete scanner instance
```

```
    public native void deleteInstance();

    // Stop decode
    public native void stopDecode();

    // Start decode
    public native void startDecode();

    // Register scan listener callbacks
    public native XCScanner onScanListener(Result v);

    // Disable all code types
    public native void disableAllSym();

    // Enable all code types
    public native void enableAllSym();

    // Enable or disable specific code type
    public native void configSymOnOff(int symid, int onoff);

    // Set max timeout of each round
    public native void setRoundTimeout(int timeMs);

    // Config decoder tag settings
    public native void configDecoderTag(int tag, int val);

    // Get value of decoder tag
    public native int getDecoderTag(int tag);


    // Set scanner workmode
    public native void setWorkMode(int workmode);

    // Get scanner workmode
    public native int getWorkMode();

    // Config scanner decode viewsize
    public native void configViewSize(int viewsize);

    // Get scanner viewsize
    public native int getViewSize();

    // Enable or disable lights control
     // This API is not to control the Lights to on/off,It is used to config
        lights controlled or uncontrolled by lower decode engine.
    public native void configLights(int lightsId, int supported);


    // Set light testmode
    // Notice: Only test function, do not use it to application
    public native void setLightTestMode(int testmode);


    // Get version of JNI
    public native String getVersion();
}
```

Interface implement sample

Description:

Sample code when we use so libary, Just call the E3Util.java interface for scanner and decode.

Java code:

```
package com.xcheng.scannere3;

import android.content.Context;
import android.graphics.SurfaceTexture;

import java.util.Arrays;
import java.util.Locale;
import java.lang.reflect.Array;

import android.hardware.camera2.CameraCaptureSession;
import android.hardware.camera2.CameraDevice;
import android.hardware.camera2.CameraManager;
import android.hardware.camera2.CaptureRequest;
import android.hardware.camera2.CameraAccessException;
import android.hardware.camera2.CameraCharacteristics;
import android.hardware.camera2.TotalCaptureResult;
import android.view.Surface;

import android.os.SystemProperties;
import android.util.Log;
import android.util.AndroidException;

import com.xcheng.scannere3.XCScanner;
public class E3Util {
    private static final String TAG = "E3Util";

    private static final int CAM_STATE_UNINITED = 0; // Camera has not inited
    private static final int CAM_STATE_OPENED = 1;   // Camera has opened
    private static final int CAM_STATE_IDLE = 2;     // Camera session created
and have no request
    private static final int CAM_STATE_PREVIEW = 3; // Camera request
processing

    private static final Object sLock = new Object();

    private static CameraCaptureSession mCameraCaptureSession = null;
    private static CameraDevice mCameraDevice = null;
    private static int mCameraStateE3 = CAM_STATE_UNINITED;
    private static boolean mAutoSetupCaptureSession = false;

    private static Surface mPreviewSurface = null;

    private static String mDecodeResult = null;
    private static String mDecodeSymbology = null;
    private static String mDecodeTime = null;
```

```
    private static boolean mIsDecoding = false;
    private static byte[] mPreviewBuf;
    private static long mTsStart = 0;
    private static long mTsStop = 0;

    // Hold this to avoid GC
    private static volatile SurfaceTexture mSurfaceTexture = null;

    private static XCScanner mScanner = null;
    private static XCScanner.Result mScannerResultListener = null;

    //Implement scanner listener
    private static final XCScanner.Result mDefScannerResultListener =
new XCScanner.Result(){
        @Override
        public void scanBeep() {
            Log.d(TAG, "mDefScannerResultListener.scanBeep");
            if (mScannerResultListener != null) {
                mScannerResultListener.scanBeep();
            }
        }

        @Override
        public void scanStart() {
            Log.d(TAG, "mDefScannerResultListener.scanStart");
            if (mScannerResultListener != null) {
                mScannerResultListener.scanStart();
            }
        }

        @Override
        public void scanResult(String sym,String content) {
            Log.d(TAG, "mDefScannerResultListener.scanResult");

            mDecodeResult = content;
            mDecodeSymbology = sym;

            if (mScannerResultListener !=null) {
                mScannerResultListener.scanResult(sym, content);
            }
        }
    };

    public E3Util() {
    }

    //Get scanner instance
    public static XCScanner getScanInstance() {
        synchronized (sLock) {
            if (mScanner == null) {
                mScanner = XCScanner.newInstance();
            }
        }
        return mScanner;
    }
```

```
    // Print CameraCharacteristics
    private static String formatCameraCharacteristics(CameraCharacteristics
info) {
        String infoText;
        if (info != null) {
            StringBuilder infoBuilder = new StringBuilder( "Camera
                    characteristics:\n\n");

            for (CameraCharacteristics.Key<?> key : info.getKeys()) {
                infoBuilder.append(String.format(Locale.US, "%s: ",
                        key.getName()));

                Object val = info.get(key);
                if (val.getClass().isArray()) {
                    // Iterate an array-type value
                    // Assumes camera characteristics won't have arrays of
 arrays as values
                    int len = Array.getLength(val);
                    infoBuilder.append("[ ");
                    for (int i = 0; i < len; i++) {
                        infoBuilder.append(String.format(Locale.US, "%s%s",
                                Array.get(val, i), (i + 1 == len) ? ""
                                        : ", "));
                    }
                    infoBuilder.append(" ]\n\n");
                } else {
                    // Single value
                    infoBuilder.append(String.format(Locale.US, "%s\n\n",
                    val.toString()));

                }
            }

            infoText = infoBuilder.toString();
        } else {
            infoText = "No info";
        }
        return infoText;
    }

    public static void setupScanResultListener(XCScanner.Result listener) {
        mScannerResultListener = listener;
    }


    private static final CameraCaptureSession.CaptureCallback mCamCaptureCb
 = null;

    // Camera Sesson Callback
    private static final CameraCaptureSession.StateCallback mCamSessionCb =
 new CameraCaptureSession.StateCallback() {
        @Override
        public void onConfigured(CameraCaptureSession session) {
             Log.d(TAG, "CameraCaptureSession::onConfigured");
             mCameraCaptureSession = session;
        }
```

```
        @Override
        public void onConfigureFailed(CameraCaptureSession session) {
            Log.d(TAG, "CameraCaptureSession::onConfigureFailed");
        }

        @Override
        public void onReady(CameraCaptureSession session) {
            Log.d(TAG, "CameraCaptureSession::onReady");

            if (mAutoSetupCaptureSession) {
                mAutoSetupCaptureSession = false;

                Log.d(TAG, "AutoSetupCaptureSession");

                try {
                    CaptureRequest.Builder builder;

                    builder =
mCameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW);
                    builder.addTarget(mPreviewSurface);

                    session.setRepeatingRequest(builder.build(), mCamCaptureCb,

null);
                } catch (CameraAccessException e1) {
                    e1.printStackTrace();
                }
            }

            mCameraStateE3 = CAM_STATE_IDLE;
        }


        @Override
        public void onActive(CameraCaptureSession session) {
            Log.d(TAG, "CameraCaptureSession::onActive");

            mCameraStateE3 = CAM_STATE_PREVIEW;
        }

        @Override
        public void onClosed(CameraCaptureSession session) {
            Log.d(TAG, "CameraCaptureSession::onClosed");
            mCameraCaptureSession = null;
            mAutoSetupCaptureSession = false;
        }
    };

    // Camera Device StateCallback
    private static final CameraDevice.StateCallback mCamDevCb =
new CameraDevice.StateCallback() {
        @Override
        public void onOpened(CameraDevice camera) {
            mCameraDevice = camera;
            Log.d(TAG, "CameraDevice::onOpened");
            mCameraStateE3 = CAM_STATE_OPENED;
```

```
            if (mPreviewSurface == null) {
               int resWidth = SystemProperties.getInt("vendor.debug.scan.resw",
844);
              int resHeight =
SystemProperties.getInt("vendor.debug.scan.resh", 640);

                if (mSurfaceTexture == null)
                {
                    mSurfaceTexture = new SurfaceTexture(0);
                }

                mSurfaceTexture.setDefaultBufferSize(resWidth, resHeight);
                mPreviewSurface = new Surface(mSurfaceTexture);
            }
            try {

mCameraDevice.createCaptureSession(Arrays.asList(mPreviewSurface),
                                                    mCamSessionCb,
                                                    null);
            } catch (CameraAccessException e) {
                e.printStackTrace();
            }
        }

        @Override
        public void onClosed(CameraDevice camera) {
            mCameraDevice = null;
            mCameraStateE3 = CAM_STATE_UNINITED;
        }

        @Override
        public void onError(CameraDevice camera, int error) {
            Log.d(TAG, "CameraDevice::onError, error = " + error);
            camera.close();
        }
        @Override
        public void onDisconnected(CameraDevice camera) {
            Log.d(TAG, "CameraDevice::onDisconnected");
            camera.close();
        }
    };

    // Open camera and start scanner
    public static long initDecoderLibrary(Context context) {
           synchronized (sLock) {

            String camid = SystemProperties.get("vendor.debug.scan.camid", "2");

             Log.d(TAG, "initDecoderLibrary+");

             if (mScanner == null) {
                mScanner = XCScanner.newInstance();
                mScanner.onScanListener(mDefScannerResultListener);
            }
```

```
           CameraManager manager = (CameraManager)
context.getSystemService(Context.CAMERA_SERVICE);

            try {
                if (mCameraDevice == null) {
                    manager.openCamera(camid, mCamDevCb, null);
                }
            } catch (CameraAccessException e) {
                e.printStackTrace();
            }
            ScanUtil.initSettings(context);
        }

        Log.d(TAG, "initDecoderLibrary-");

        return 0;
    }

    // Stop camera
    public static void releaseDecodeLibrary() {
        Log.d(TAG, "releaseDecodeLibrary+");
        mDecodeResult = null;
        mDecodeSymbology = null;
        mDecodeTime = null;

        if (mCameraCaptureSession != null)
        {
            mCameraCaptureSession.close();
        }
        mCameraCaptureSession = null;

        if (mCameraDevice != null)
        {
            mCameraDevice.close();
        }
        mCameraDevice = null;

        if (mPreviewSurface != null)
        {
            mPreviewSurface.release();
        }
        mPreviewSurface = null;

       if (mSurfaceTexture != null) {
            mSurfaceTexture.release();
        }
        mSurfaceTexture = null;

        if (mScanner != null) {
            if (mIsDecoding) {
                mScanner.stopDecode();
            }
            mScanner.deleteInstance();
            mScanner = null;
        }
```

```
        mIsDecoding = false;

        Log.d(TAG, "releaseDecodeLibrary-");
    }

    //Do decode
    public static void doDecode() {
        Log.d(TAG, "doDecode+");
        mTsStart = System.currentTimeMillis();

        if (mCameraCaptureSession != null && mCameraDevice != null)
            { if (mCameraStateE3 == CAM_STATE_IDLE) {
                mAutoSetupCaptureSession = false;
                Log.d(TAG, "createCaptureRequest");
                try {
                    CaptureRequest.Builder builder;
                    builder =
mCameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW);
                    builder.addTarget(mPreviewSurface);
                    mCameraCaptureSession.setRepeatingRequest(builder.build(),
mCamCaptureCb, null);
                } catch (CameraAccessException e1) {
                    e1.printStackTrace();
                }
            } else if (mCameraStateE3 == CAM_STATE_OPENED) {
                // Create capture while camera opened
                mAutoSetupCaptureSession = true;
            } else if (mCameraStateE3 == CAM_STATE_PREVIEW) {
                // Camera is previewing, not need createCaptureRequest
            }
        } else {
            // Create capture while camera opened
            mAutoSetupCaptureSession = true;
        }

        if (mScanner != null) {
            mScanner.startDecode();
        }

        mDecodeResult = null;
        mDecodeSymbology = null;
        mIsDecoding = true;

        Log.d(TAG, "doDecode-");
    }
```

```
//Stop decode
public static void stopDecode(){
    Log.d(TAG, "stopDecode+");

    if (mIsDecoding) {
        if (mCameraCaptureSession != null && mCameraDevice != null) {
            Log.d(TAG, "stopRepeating");

            if (mCameraStateE3 == CAM_STATE_PREVIEW) {
                try {
                    mCameraCaptureSession.stopRepeating();
                } catch (AndroidException e1) {
                    e1.printStackTrace();
                }
            }
        }

        if (mScanner != null) {
            mScanner.stopDecode();
        }
    }

    //mDecodeResult = null;
    //mDecodeSymbology = null;

    mDecodeTime = null;
    mIsDecoding = false;

    Log.d(TAG, "stopDecode-");
}

//Is or not decoding
public static boolean IsDecoding() {
    return mIsDecoding;
}

//Get decode result data
public static String getDecodeResult(boolean onceOnly) {
    Log.d(TAG, "getDecodeResult+");
    Log.d(TAG, "mDecodeResult = " + mDecodeResult);
    Log.d(TAG, "getDecodeResult-");
    String result = mDecodeResult;
    if (onceOnly) {
        mDecodeResult = null;
    }
    return result;
}

//Get decode symbology
public static String getDecodeSymbology(boolean onceOnly) {
    Log.d(TAG, "getDecodeSymbology+");
    Log.d(TAG, "mDecodeSymbology = " + mDecodeSymbology);
```

```
        Log.d(TAG, "getDecodeSymbology-");
        String sym = mDecodeSymbology;
        if (onceOnly) {
            mDecodeSymbology = null;
        }
        return sym;
    }

    //Set scanner mode
    public static void setScanMode(int workmode) {
        Log.d(TAG, "setScanMode+ workmode = " + workmode);

        if (mScanner != null) {
            if (workmode >= 0) {
                mScanner.setWorkMode(workmode);
            }
        }

        Log.d(TAG, "setScanMode-");
    }
    // Get scan mode
    public static int getScanMode() {
       Log.d(TAG, "getScanMode+");
       int mode = XCScanner.WORKMODE_SINGLESHOT;

        if (mScanner != null) {
            mode = mScanner.getWorkMode();
        }

        Log.d(TAG, "getScanMode-, mode = " + mode);
        return mode;
    }

    public static void configViewSize(int viewsize) {
        Log.d(TAG, "configViewSize+ viewsize = " + viewsize);

        if (mScanner != null) {
            if (viewsize >= 0) {
                mScanner.configViewSize(viewsize);
            }
        }

        Log.d(TAG, "configViewSize-");
    }

    // Get viewsize
    public static int getViewSize() {
        Log.d(TAG, "getViewSize+");
        int viewsize = XCScanner.VIEWSIZE_100;

        if (mScanner != null) {
            viewsize = mScanner.getViewSize();
        }

        Log.d(TAG, "getViewSize-, viewsize = " + viewsize);
        return viewsize;
    }
```

```
// Set decode timeout
public static void setTimeout(int timeout_ms) {
    Log.d(TAG, "setTimeout+ timeout = " + timeout_ms);

    if (mScanner != null) {
        if (timeout_ms > 0) {
            mScanner.setRoundTimeout(timeout_ms);
        }
    }

    Log.d(TAG, "setTimeout-");
}

// Enable all code types
public static void enableAllCodeTypes() {
    Log.d(TAG, "enableAllCodeTypes+");
    if (mScanner != null) {
        mScanner.enableAllSym();
    }
    Log.d(TAG, "enableAllCodeTypes-");
}

// Disable all code types
public static void disableAllCodeTypes() {
    Log.d(TAG, "disableAllCodeTypes+");
    if (mScanner != null) {
        mScanner.disableAllSym();
    }
    Log.d(TAG, "disableAllCodeTypes-");
}

// Config specific code type
public static void setCodeTypeOnOff(String symbology, boolean enable) {
    Log.d(TAG, "setCodeTypeOnOff+");

    if (mScanner == null) {
        Log.e(TAG, "mScanner null");

        return;
    }

    int paramID , tag_val;
     switch(getSymID(symbology)) {
         case XCScanner.E3_SYM_QRCODE:
                 paramID = XCScanner.TAG_QR_ENABLED;
                 tag_val = getNumParameter(paramID);
                 if (enable) {
                     tag_val |= (0x01 << 0);
                     tag_val |= (0x01 << 2);
                     tag_val |= (0x01 << 4);
                 } else {
                     tag_val &= (~(0x01 << 0));
                     tag_val &= (~(0x01 << 2));
                     tag_val &= (~(0x01 << 4));
                 }
                 setParameter(paramID, tag_val);
             break;
```

```
            case XCScanner.E3_SYM_DATAMATRIX:
                    paramID = XCScanner.TAG_DATAMATRIX_ENABLED;
                    tag_val = getNumParameter(paramID);
                    if (enable) {
                        tag_val |= (0x01 << 0);
                    } else {
                        tag_val &= (~(0x01 << 0));
                    }
                    setParameter(paramID, tag_val);
                break;
           case XCScanner.E3_SYM_GS1_DATABAR:
                       setDecoderTag(XCScanner.TAG_RSS_14_ENABLED, enable ? 1 :
0);
                       setDecoderTag(XCScanner.TAG_RSS_EXPANDED_ENABLED,
enable ? 1 : 0);
                       setDecoderTag(XCScanner.TAG_RSS_LIMITED_ENABLED,
enable ? 1 : 0);
                 break;
           default:
                       mScanner.configSymOnOff(getSymID(symbology), enable ?
1 : 0);
                break;
    }

    public static int getSymID(String symbology){

        int symid = -1;

        // Convert Application symbol string to Native symbol string
        if (symbology.equals("AZTEC")) {
            symid = XCScanner.E3_SYM_AZTEC;
        } else if (symbology.equals("C11")) {
            symid = XCScanner.E3_SYM_CODE11;
        } else if (symbology.equals("C39")) {
           symid = XCScanner.E3_SYM_CODE39;
        } else if (symbology.equals("C93")) {
           symid = XCScanner.E3_SYM_CODE93;
        } else if (symbology.equals("C128")) {
           symid = XCScanner.E3_SYM_CODE128;
        } else if (symbology.equals("Codabar")) {
            symid = XCScanner.E3_SYM_CODABAR;
        } else if (symbology.equals("CODEBLOCK F")) {
            symid = XCScanner.E3_SYM_CODABLOCK_F;
        } else if (symbology.equals("DATA MATRIX")) {
            symid = XCScanner.E3_SYM_DATAMATRIX;
        } else if (symbology.equals("EAN-8")) {
            symid = XCScanner.E3_SYM_EAN8;
        } else if (symbology.equals("EAN-13")) {
            symid = XCScanner.E3_SYM_EAN13;
```

```
    } else if (symbology.equals("GS1 DATABAR")) {
        symid = XCScanner.E3_SYM_GS1_DATABAR;
    } else if (symbology.equals("HAXIN")) {
        symid = XCScanner.E3_SYM_HANXIN;
    } else if (symbology.equals("HK25")) {
        symid = XCScanner.E3_SYM_HK25;
    } else if (symbology.equals("I25")) {
         symid = XCScanner.E3_SYM_ITF25;
    } else if (symbology.equals("MATRIX 25")) {
        symid = XCScanner.E3_SYM_MATRIX25;
    } else if (symbology.equals("MAXICODE")) {
        symid = XCScanner.E3_SYM_MAXICODE;
    } else if (symbology.equals("MSI")) {
        symid = XCScanner.E3_SYM_MSI;
    } else if (symbology.equals("MICROPDF")) {
         symid = XCScanner.E3_SYM_MICROPDF417;
    } else if (symbology.equals("PDF417")) {
        symid = XCScanner.E3_SYM_PDF417;
    } else if (symbology.equals("POSTAL")) {
         symid = XCScanner.E3_SYM_USPS_4_STATE;
    } else if (symbology.equals("QR CODE")) {
        symid = XCScanner.E3_SYM_QRCODE;
    } else if (symbology.equals("STRAIGHT 25")) {
         symid = XCScanner.E3_SYM_INDUSTRIAL25;
    } else if (symbology.equals("TELEPEN")) {
        symid = XCScanner.E3_SYM_TELEPEN;
    } else if (symbology.equals("TRIOPTIC")) {
        symid = XCScanner.E3_SYM_TRIOPTIC;
    } else if (symbology.equals("UPC-A")) {
        symid = XCScanner.E3_SYM_UPCA;
    } else if (symbology.equals("UPC-E")) {
        symid = XCScanner.E3_SYM_UPCE;
    }else if (symbology.equals("GS1-128")) {
        symid = XCScanner.E3_SYM_GS1_128;
    } else if (symbology.equals("GS1 DataBar Limited")) {
        symid = XCScanner.E3_SYM_GS1_LIMITED;
    }else if (symbology.equals("GS1 DataBar Expanded")) {
        symid = XCScanner.E3_SYM_GS1_EXPANDED;
    }
    return symid;
}

public static int getNumParameter(int paramNum) {
    return getDecoderTag(paramNum);
}

public static int setParameter(int paramNum, int paramVal) {
    return setDecoderTag(paramNum, paramVal);
}

public static int setDecoderTag(int tag, int value) {
    int ret = 0;
    if (mScanner == null) {
        ret = -1;
```

```
        } else {
            mScanner.configDecoderTag(tag, value);
        }

        return ret;
    }

    public static int getDecoderTag(int tag) {
        int ret = 0;
        if (mScanner == null) {
            ret = -1;
        } else {
            ret = mScanner.getDecoderTag(tag);
        }

        return ret;
    }

    public static void setLightAimSupport(boolean suppoted) {
        Log.d(TAG, "setLightAimSupport+");
        if (mScanner != null) {
            if (suppoted) {
                mScanner.configLights(XCScanner.LIGHTS_ID_AIM, 1);
            } else {
                mScanner.configLights(XCScanner.LIGHTS_ID_AIM, 0);
            }
        }
        Log.d(TAG, "setLightAimSupport-");
    }

    public static void setLightFlashSupport(boolean suppoted) {
        Log.d(TAG, "setLightFlashSupport+");
        if (mScanner != null) {
            if (suppoted) {
                mScanner.configLights(XCScanner.LIGHTS_ID_FLASH, 1);
            } else {
                mScanner.configLights(XCScanner.LIGHTS_ID_FLASH, 0);
            }
        }
        Log.d(TAG, "setLightFlashSupport-");
    }

    public static void setLightTestMode(int testMode) {
        Log.d(TAG, "setLightTestMode+ testMode: " + testMode);
        if (mScanner != null) {
            mScanner.setLightTestMode(testMode);
        }
        Log.d(TAG, "setLightTestMode-");
    };
}
```

Scanner intent

To open and close scanner device intent as follow code

```
public static final String ACTION_OPEN_SCAN_BROADCAST =
"com.xcheng.scanner.action.OPEN_SCAN_BROADCAST";
public static final String ACTION_CLOSE_SCAN_BROADCAST =
"com.xcheng.scanner.action.CLOSE_SCAN_BROADCAST";

public static void openCloseDevice(Context context, boolean
    isOpen){ Intent intent = new Intent();
    if (isOpen) {
        intent.setAction(ACTION_OPEN_SCAN_BROADCAST);//open scanner
    } else {
        intent.setAction(ACTION_CLOSE_SCAN_BROADCAST);//close scanner
    }
    context.sendBroadcast(intent);
}
```

To control code scanning button switch

```
public static final String ACTION_CONTROL_SCANKEY =
 "com.xcheng.scanner.action.ACTION_CONTROL_SCANKEY";
public static final String EXTRA_SCANKEY_CODE = "extra_scankey_code";
public static final String EXTRA_SCANKEY_STATUS = "extra_scankey_STATUS";
public static final int LEFT_SCANNER_KEYCODE = 280;
public static final int RIGHT_SCANNER_KEYCODE = 281;
public static final int FRONT_SCANNER_KEYCODE = 141;

public static void scannerButtonControl(Context context, int keyCode, boolean
isOpen){
    Intent intent = new
    Intent(ACTION_CONTROL_SCANKEY);
    intent.putExtra(EXTRA_SCANKEY_CODE, keyCode);
    intent.putExtra(EXTRA_SCANKEY_STATUS, isOpen);
    context.sendBroadcast(intent);
}
```

To listener scanner result data as follow code

```
//This code copy from E3Util.java
private static final XCScanner.Result mDefScannerResultListener =
new XCScanner.Result(){
```

```
        @Override
        public void scanBeep()
            if (mScannerResultListener != null) {
                mScannerResultListener.scanBeep();
            }
        }

        @Override
        public void scanStart() {
            if (mScannerResultListener != null) {
                mScannerResultListener.scanStart();
            }
        }

        @Override
        public void scanResult(String sym,String content)
            { mDecodeResult = content;
            mDecodeSymbology = sym;

            if (mScannerResultListener != null)
                { mScannerResultListener.scanResult(sym, content);//Here get
scanner result info, symbology and content
            }
        }
    };
```

To set symbology enable and disable

```
public static final String ACTION_ENABLE_TYPE =
"com.xcheng.scanner.action.ENABLE_SCANTYPE_BROADCAST";
public static final String ACTION_DISABLE_TYPE =
"com.xcheng.scanner.action.DISABLE_SCANTYPE_BROADCAST";
public static final String SCANTYPE = "scantype";

public void enableOrDisableSymbology(int symbology, bool enabled)
     { Intent intent = new Intent();
     if (enabled) {
        intent.SetAction(ACTION_ENABLE_TYPE);
        intent.PutExtra(SCANTYPE, symbology);
     }
     else {
         intent.SetAction(ACTION_DISABLE_TYPE);
         intent.PutExtra(SCANTYPE, symbology);
     }

     context.SendBroadcast(intent);
 }
```

To set result response method as follow code

```
public static final String ACTION_CONTROL_DATA_EVENT
= "com.xcheng.scanner.action.CONTROL_DATA_EVENT";
public static final String EXTRA_DATA_EVENT = "extra_data_event";
public static final string DATA_RECV_NONE = "NONE";
public static final string DATA_RECV_BROADCAST_EVENT = "BROADCAST_EVENT";
public static final string DATA_RECV_KEYBOARD_EVENT = "KEYBOARD_EVENT";

public void setScanResultRspMethodBroadCast() {
    Intent intent = new Intent();
    intent.SetAction(ACTION_CONTROL_DATA_EVENT);
    intent.PutExtra(EXTRA_DATA_EVENT, DATA_RECV_BROADCAST_EVENT);
    context.SendBroadcast(intent);
}
```

To load and save setting

```
public static final String ACTION_LOAD_SETTINGS
= "com.xcheng.scanner.action.LOAD_SETTINGS";
public static final String ACTION_SAVE_SETTINGS
= "com.xcheng.scanner.action.SAVE_SETTINGS";
public final static String XML_FILE_NAME =
"com.xcheng.scannere3_preferences.xml";

public static void loadSettings(Context context) {
    loadSettings(context, getSdCardPath() + XML_FILE_NAME);
}

private static void loadSettings(Context context, String fileName) {
    if (!isFileExist(fileName)) {
        return;
    }
    SharedPreferences sharedPreferences = getSharedPreferences(context);
    Editor editor = sharedPreferences.edit();

    try {
        XmlPullParser pullParser =
        Xml.newPullParser(); InputStream is = new
ByteArrayInputStream(readFileData(fileName).getBytes("UTF-
        8")); pullParser.setInput(is, "utf-8");
        int eventType = pullParser.getEventType();
        while (eventType != XmlPullParser.END_DOCUMENT) {
           String tagName = pullParser.getName();

            // Parse "string" items
            if (!TextUtils.isEmpty(tagName) && "string".equals(tagName))
                { if (eventType == XmlPullParser.START_TAG) {
                   String nameString = pullParser.getAttributeValue(null,
"name");
                    String valueString = pullParser.nextText();
                    if (!TextUtils.isEmpty(valueString)) {
                        editor.putString(nameString, valueString);
                        editor.commit();
                    }
                }
            }
```

```
            // Parse "boolean" items
            if (!TextUtils.isEmpty(tagName) && "boolean".equals(tagName))
                { if (eventType == XmlPullParser.START_TAG) {
                    String nameBoolean = pullParser.getAttributeValue(null,
"name");
                    String valueBoolean = pullParser.getAttributeValue(null,
"value");
                   if (!TextUtils.isEmpty(valueBoolean)) {
                        boolean enabled = Boolean.parseBoolean(valueBoolean);
                        editor.putBoolean(nameBoolean, enabled);
                        editor.commit();

                    }
                }
            }

            // Parse "int" items
            if (!TextUtils.isEmpty(tagName) && "int".equals(tagName)){
                if (eventType == XmlPullParser.START_TAG) {
                    String nameInt = pullParser.getAttributeValue(null, "name");
                    String valueInt = pullParser.getAttributeValue(null,
"value");
                    if (!TextUtils.isEmpty(valueInt)) {
                        editor.putInt(nameInt, Integer.parseInt(valueInt));
                        editor.commit();
                    }
                }
            }
            eventType = pullParser.next();
        }

        // Config timeout value String maxTimeoutString =
sharedPreferences.getString("pref_max_scan_time","2000");
        if (isValidTimeout(context, maxTimeoutString)) {
            E3Util.setTimeout(Integer.parseInt(maxTimeoutString));
        }

        // Apply all code format settings
        applyAllModulesSetting(context);
    } catch (Exception e){
        e.printStackTrace();
    }
}

public static void saveSettings(Context context) {
    String packageName = context.getPackageName();
    writeFileDataToSd(getSdCardPath() + XML_FILE_NAME, /data/data/" +
    packageName + "/shared_prefs/" + XML_FILE_NAME);
}

public static void writeFileDataToSd(String toFileName, String fromFileName) {
    try {
        File file = new File(toFileName);
        FileOutputStream fos = new FileOutputStream(file);
        fos.write(readFileData(fromFileName).getBytes());
```

```
        fos.flush();
        fos.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static boolean isFileExist(String fileName) {
    boolean isExist = false;
    try {
        File file = new File(fileName);
        if (file.exists()) {
            isExist = true;
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return isExist  ;
}

public static SharedPreferences getSharedPreferences(Context context){
    return context.getSharedPreferences("com.xcheng.scannere3_preferences",
 Context.MODE_PRIVATE);
}

public static String readFileData(String fileName) {
    String result = "";
    try {
        File file = new File(fileName);
        FileInputStream fis = new
        FileInputStream(file); int length =
        fis.available(); byte[] buffer = new
        byte[length];
        fis.read(buffer);
        result = new String(buffer, "UTF-8");
    } catch (Exception e) {
         e.printStackTrace();
    }
    return result;
}

public static boolean isValidTimeout(Context context, String timeout) {
    String[] timeoutValues =
 context.getResources().getStringArray(R.array.scan_timeout_values);
    for (int i = 0; i < timeoutValues.length; i++) {
        if (timeoutValues[i].equals(timeout)) {
            return true;
        }
    }
    return false;
}
```

```
// Apply all code format support settings
// If switch_all true, not need to apply detailed settings
// If switch_all false, need to enable code format of AllModulesSetting
enabled items.
public static void applyAllModulesSetting(Context context) {
    SharedPreferences preferences = getSharedPreferences(context);

    // Config switch_all value
    boolean swtich_all_on = preferences.getBoolean("swtich_all", true);
    if (swtich_all_on) {
        E3Util.enableAllCodeTypes();
    } else {
        E3Util.disableAllCodeTypes();
    }

    // Config all configs if switch_all is off
    if (!swtich_all_on) {
        int i = 0;
        for (i = 0; i < SP_KEYS.length; i++) {
            boolean symonoff = preferences.getBoolean(SP_KEYS[i], false);

            if (symonoff) {
                E3Util.setCodeTypeOnOff(context.getString(SP_TITLE_RES[i]),
symonoff);
            }
        }
    }
```