cmbSDK for Android (v2.0.x)

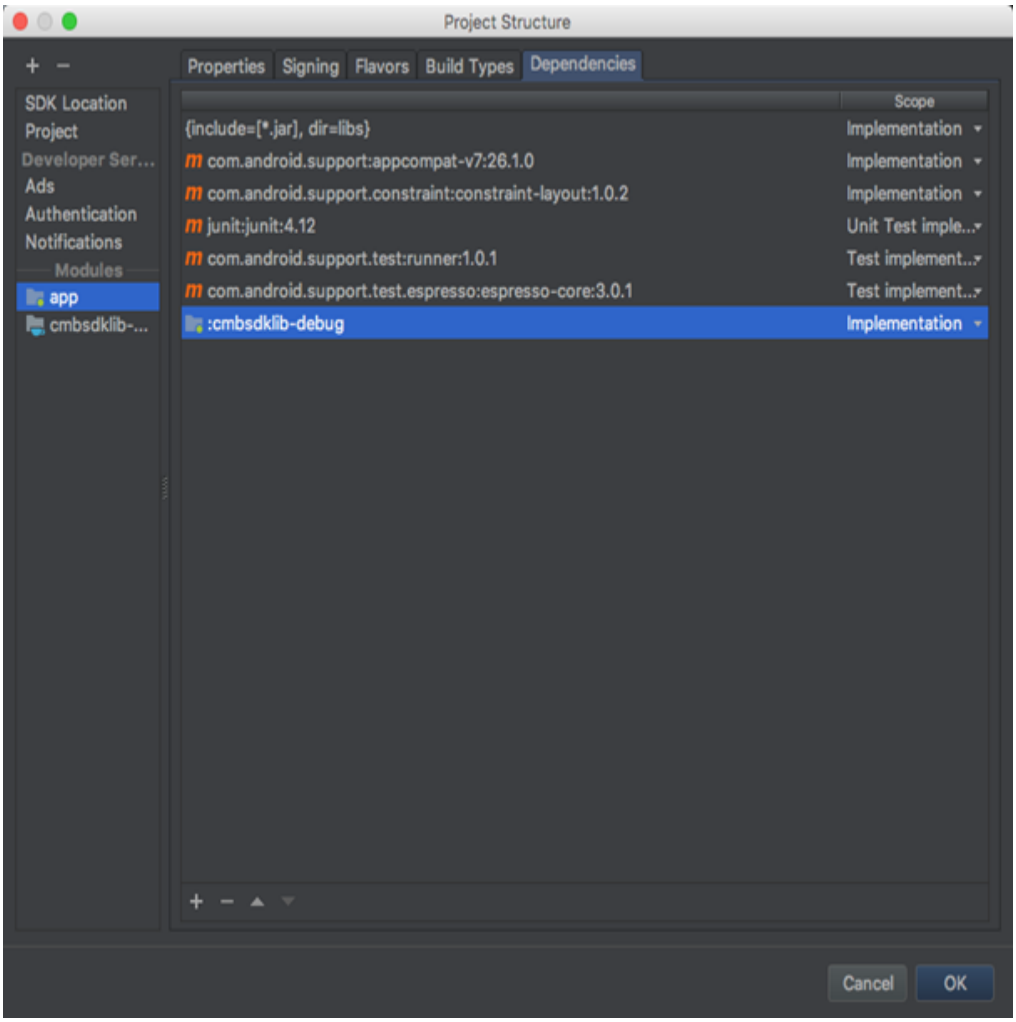
Getting Started

Note: cmbSDK is compatible with Android Studio.

Perform the following steps to install the Android cmbSDK:

1. Download the Cognex Mobile Barcode SDK for Android.
2. Start Android Studio and add SDK AAR file as module to your project:

    - Right click your app module, select **New > Module > Import .JAR/.AAR Package**, and click **Next**.
    - Browse the SDK .AAR file in the File name field and click **Finish**.
    - After the new module is available, right click your app module again, select **Open Module Settings**, and choose **Dependencies** tab.
    - Click the **+** sign on the bottom of the dialog and select **3 Module dependency**.
    - Select cmbsdklib from the popup and click **OK**.
    - The cmbsdklib module should be available now under the **Dependencies** tab.



3. To communicate with MX readers, install MX Connect app from Play Store on your mobile device.

Note: Please uninstall all other applications that connect to the MX reader with previous versions of the SDK. Update the QuickSetup application to latest version.

Licensing the SDK

If you plan to use the cmbSDK to do mobile scanning with a smartphone or tablet (with no MX mobile terminal), then the SDK requires the installation of a license key. Without a license key, the SDK will still operate, although scanned results will be obfuscated (the SDK will randomly replace characters in the scan result with an asterisk character).

Contact your Cognex Sales Representative for information on how to obtain a license key including trial licenses which can be used for 30 days to evaluate the SDK.

After obtaining your license key, add the following line in your application's AndroidManifest.xml file, under the application tag:

```
<meta-data android:name="MX_MOBILE_LICENSE" android:value="YOUR_MX_MOBILE_LICENSE"/>
```

Next, put your key in place of YOUR_MX_MOBILE_LICENSE.

```
<application
android:allowBackup="true"
android:icon="@drawable/ic_launcher"
android:label="@string/app_name"
android:theme="@style/AppTheme" >
<activity
android:name=".ScannerActivity"
android:label="@string/app_name" >
<intent-filter>
<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>

<meta-data
android:name="MX_MOBILE_LICENSE"
android:value="g/9ytJzcja+sxt4DTEDxR4hp6sZh9bmL97vUx+EE9uY=" />

</application>
```

Migrating from a Previous DataMan SDK for MX Readers

Previous SDK versions accessed the MX mobile terminal via direct USB Device or USB Accessory connection. These methods are now deprecated, and you should connect to an MX mobile terminal using the MXConnect application. (See Step 4 .)

The cmbSDK provides an easy factory method, DataManSystem.createDataManSystemForMXDevice() to create a DataManSystem for the MX mobile terminal over the MXConnect application.

Please remove any DataManSystem.createDataManSystemOverUsb() and DataManSystem.createDataManSystemOverUsbAccessory() methods from your project, and also remove the USB_ DEVICE-ATTACHED and USB_ACCESSORY_ATTACHED Intent filters and meta-data from the AndroidManifest.xml file.

You can also delete the USB and accessory descriptor xml files from the XML folder. You can also use and migrate to Barcode SDK to access the MX mobile terminal.

Writing a Mobile Application

The cmbSDK has been designed to provide a high-level, abstract interface for supported scanning devices. This includes not only the MX series of mobile terminals, but also for applications that intend to use the mobile device camera as the imaging device. The intricacies of communicating with and managing these devices is encapsulated within the SDK itself: leaving the application to just connect to the device of choice, then using it.

The primary interface between your application and a supported barcode scanning device is the ReaderDevice class. This class represents the abstraction layer to the device itself, handling all communication as well as any necessary hardware management (e.g. for smartphone scanning).

Perform the following steps to use the cmbSDK:

1. Initialize a Reader Device for the type of device you want to use (MX reader or camera reader).
2. Connect the Reader Device.
3. Configure the reader (if necessary).
4. Start scanning.

Initialization, connection, and configuration generally need to be performed only once in your application except for the following cases:

- An MX reader may become disconnected (times out from disuse,dead battery, etc.). A method has been provided to handle this case, and is discussed in a later section.
- Your application has been designed to allow the user to change devices.The cmbSDK is explicitly designed to support this: your application simply disconnects from the current device and establishes a new connection to a different device. The provided sample application has been written to explicitly demonstrate this capability.

Setting up your Application to use the Cognex Mobile Barcode SDK for Android

Perform the following steps to set up and start using the cmbSDK:

1. Import the following package members:

```
import com.cognex.dataman.sdk.CameraMode;
import com.cognex.dataman.sdk.ConnectionState;
import com.cognex.dataman.sdk.PreviewOption;
import com.cognex.mobile.barcode.sdk.ReadResult;
import com.cognex.mobile.barcode.sdk.ReadResults;
import com.cognex.mobile.barcode.sdk.ReaderDevice;
import com.cognex.mobile.barcode.sdk.ReaderDevice.Availability;
import com.cognex.mobile.barcode.sdk.ReaderDevice.OnConnectionCompletedListener;
import com.cognex.mobile.barcode.sdk.ReaderDevice.ReaderDeviceListener;
import com.cognex.mobile.barcode.sdk.ReaderDevice.Symbology;
```

2. Provide needed UI elements:

- **ViewGroup** container (like **RelativeLayout**) for the camera preview
- **ImageView** nested inside the**ViewGroup** container with matching size as its parent for showing the last frame of a preview or scanning session:

The below examples use the following:

```
private RelativeLayout rlPreviewContainer;
private ImageView ivPreview;
```

3. The following interfaces are necessary to monitor the connection state of the reader and receive information about the read code:

```
public class ScannerActivity extends Activity implements OnConnectionCompletedListener, ReaderDeviceListener {
....
@Override
public void onConnectionCompleted(ReaderDevice reader, Throwable error) {
if (error != null) {
// READER DISCONNECTED (ERROR OCCURED)
} }
@Override
public void onConnectionStateChanged(ReaderDevice reader) {
if (reader.getConnectionState() == ConnectionState.Connected) { // READER CONNECTED
} else if (reader.getConnectionState() == ConnectionState.Disconnected) { // READER DISCONNECTED
} }
@Override
public void onReadResultReceived(ReaderDevice reader, ReadResults results) {
if (results.getCount() > 0) {
ReadResult result = results.getResultAt(0);
// USE String symbologyName; String code; Bitmap frame; VARIABLES IN YOUR APPLICATION
if (result.isGoodRead()) {
String symbologyName;
String code = result.getReadString();
Symbology symbology = result.getSymbology(); if (symbology != null) {
symbologyName = symbology.getName();
tvSymbology.setText(symbologyName); } else {
tvSymbology.setText("UNKNOWN SYMBOLOGY"); }
tvCode.setText(code); } else {
tvSymbology.setText("NO READ");
tvCode.setText(""); }
Bitmap frame = result.getImage();
ivPreview.setImageBitmap(frame); }
// READY TO SCAN AGAIN }
@Override
public void onAvailabilityChanged(ReaderDevice reader) {
if (reader.getAvailability() == Availability.AVAILABLE) { // READER DEVICE IS AVAILABLE AND CAN BE CONNECTED
} else {
// DISCONNECT DEVICE
} }
.... }
```

4. Instantiate a ReaderDevice object.

The cmbSDK provides two different reader class initializers: one for scanning using an MX mobile terminal (like the MX- 1000 or MX-1502) and another for scanning using the built-in camera of the mobile device (Android Phones, Android Tablets, etc.).

Using the MX Reader

Initialize a Reader Device object for MX readers using the following factory method:

```
boolean listeningForUSB = false;

ScannerActivity.readerDevice = ReaderDevice.getMXDevice(ScannerActivity.this); if (!listeningForUSB) {

readerDevice.startAvailabilityListening();
listeningForUSB = true;
}
```

The availability of the MX mobile terminal can change when the device turns ON or OFF, or if the USB cable gets connected or disconnected. You can handle those changes using the following ReaderDeviceListener interface method (implemented in Step 3 above):

```
public void onAvailabilityChanged(ReaderDevice reader);
```

Using the Camera Reader

Barcode scanning with the built-in camera of the mobile device can be more complex than with an MX mobile terminal. The cmbSDK supports several configurations to provide the maximum flexibility. This includes the support of optional, external aimers/illumination, as well as the ability to customize the appearance of the live-stream preview.

To scan barcodes using the built-in camera of the mobile device, initialize the ReaderDevice object using the getPhoneCameraDevice static method. The camera reader has several options when initialized. The following parameters are required:

- Context
- CameraMode
- PreviewOption
- ViewGroup

The Context parameter simply provides a reference to the activity you are currently in.
The CameraMode parameter is of type CameraMode (defined in **CameraMode.java**) and it accepts one of the following values:

- **NO_AIMER**: This initializes the reader to use a live-stream preview (on the mobile device screen)so the user can position the barcode within the camera's field of view for detection and decoding. Use this mode when the mobile device does not have an aiming accessory.
- **PASSIVE_AIMER**: This initializes the reader to usea passive aimer, which is an accessory that is attached to the mobile device or mobile device case that uses the built-in LED flash of the mobile device as a light source for projecting an aiming pattern. In this mode, no live-stream preview is presented on the device screen, since an aiming pattern will be projected.
- **FRONT_CAMERA**: This initializes the reader to use the mobile front facing camera of the device, if available (not all mobile devices have a front camera). This is an unusual, but possible configuration. Most front facing cameras do not have auto focus and illumination, and provide significantly lower resolution images. This option should be used with care. In this mode, illumination is not available.

All of the above modes provide the following default settings for the reader:

- The rea rcamera is used.
- The zoom feature is available and a buttonto control it is visible on the live-stream preview (if displayed).
- The simulated hardware trigger is disabled.
- When start Scanning( ) is called,the decoding process is started. (See PreviewOption.PAUSED below for more details).

Based on the selected mode, the following additional options and behaviors are set:

- **NO_AIMER** (NoAimer)
- The live-stream preview is displayed when the *startScanning()* method is called.
- Illumination is available, and a button to control it is visible on the live-stream preview.
- If commands are sent to the reader for aimer control, they will be ignored.

- **PASSIVE_AIMER** (Passive Aimer)
- The live-stream preview will not be displayed when the *startScanning()* method is called.
- Illumination is not available, and the live-stream preview will not have an illumination button.
- If commands are sent to the reader for illumination control, they will be ignored, since it is assumed in this mode that the built-in LED of the mobile device is being used for the aimer.

- **FRONT_CAMERA** (FrontCamera)
- The live-stream preview is displayed when the startScanning() method is called.
- The front camera is used.
- Illumination is not available and the live-stream preview will not have an illumination button.
- If commands are sent to the reader for aimer or illumination control, they will be ignored.

The *PreviewOption* parameter is of type *PreviewOption* (defined in **PreviewOption.java**), and is used to change the reader's default values or override defaults derived from the selected *CameraMode*. Multiple options can be specified by OR-ing them when passing the parameter. The available options are:

- **DEFAULTS**: Use this option to accept all defaults set by the CameraMode.
- **NO_ZOOM_BUTTON**: This hides the zoom button on the live-stream preview, preventing a user from adjusting the mobile device camera's zoom.

- **NO_ILLUMINATION_BUTTON**: This hides the illumination button on the live-stream preview, preventing a user from toggling the illumination.
- **HARDWARE_TRIGGER**: This enables a simulated hardware trigger (the volume down button) for starting scanning on the mobile device. This button only starts scanning when pressed. Iit does not need to be held like a more traditional purpose-built scanner's trigger. Pressing the button a second time does not stop the scanning process.
- **PAUSED**: If using a live-stream preview, when this option is set, the preview will be displayed when the *startScanning()* method is called, but the reader will not started decoding (i.e. looking for barcodes) until the user presses the on-screen scanning button to actually start the scanning process.
- **ALWAYS_SHOW**: This option forces a live-stream preview to be displayed, even if an aiming mode has been selected (e.g. CameraMode == PASSIVE_AIMER)

The last parameter of the ViewGroup type specifies the container for the live-stream preview.

Examples

Create a reader with no aimer, no zoom button, and using a soft trigger:

```
ScannerActivity.readerDevice = ReaderDevice.getPhoneCameraDevice( ScannerActivity.this,
CameraMode.NO_AIMER,
PreviewOption.NO_ZOOM_BUTTON | PreviewOption.PAUSED, rlPreviewContainer);
```

This starts a preview with the scanner paused and a soft trigger button to toggle scanning. After pressing the soft trigger button, the rlPreviewContainer should look like this:

The viewfinder in the above image has an active scanning surface, a result of having set active symbologies. For more details on this topic, see Configuring the Reader Device).

Requesting Camera permission for Phone Camera Scanner

From Android 6.0 and above you need to request permission from the user to access the phone camera.
If the phone camera cannot be opened due to permission issues the *onConnectionCompleted(readerDevice, error)* callback contains a *CameraPermissionException* in the error parameter.
You can check for this exception type with the *instanceof* operator and request permission within the Activity.

```
if (error instanceof CameraPermissionException)
      ActivityCompat.requestPermissions(((ScannerActivity) this), new String[]{Manifest.permission.CAMERA}, REQUEST_PERMISS
```

Please note, that you need to implement *ActivityCompat.OnRequestPermissionResultCallback* interface in your Activity to catch user permission result.
To handle user response in *onRequestPermissionResult(...)*, you may use the following code to retry connecting to the PhoneCamera.

```
@Override
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {
    if (requestCode == REQUEST_PERMISSION_CODE) {
```

```
        if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            if (readerDevice != null && readerDevice.getConnectionState() != ConnectionState.Connected)
                readerDevice.connect(ScannerActivity.this);
        } else {
            if (ActivityCompat.shouldShowRequestPermissionRationale(((ScannerActivity) this), Manifest.permission.CAMERA))
                AlertDialog.Builder builder = new AlertDialog.Builder(this)
                        .setMessage("You need to allow access to the Camera")
                        .setPositiveButton("OK", new DialogInterface.OnClickListener() {
                            @Override
                            public void onClick(
                                    DialogInterface dialogInterface,
                                    int i) {
                                ActivityCompat.requestPermissions(ScannerActivity.this, new String[]{Manifest.permission.CA
                            }
                        })
                        .setNegativeButton("Cancel", null);
                AlertDialog dialog = builder.create();
                dialog.show();
            }
        }
    }
}
```

Connecting to the Device

Before connecting, set the ReaderDeviceListener object, in order to receive events:

```
readerDevice.setReaderDeviceListener(ScannerActivity.this);
```

See Step 3 for details.

Additionally, you can enable sending the last triggered image from the reader:

```
readerDevice.enableImage(true);
```

After initializing the ReaderDevice and setting a listener method to handle responses from the reader, the connect method can be invoked, which takes a OnConnectionCompletedListener (see Step 3 for details) as parameter:

```
 //Make sure the device is turned ON and ready
readerDevice.connect(ScannerActivity.this);
```

If everything was done correctly, the following listener methods will be called with the new ReaderDevice status information. The onConnectionCompleted method (passed as parameter of connect) will also be invoked as the connection process completes, providing a Throwable object, if there was a connection error.

```
 public void onConnectionStateChanged(ReaderDevice reader);
public void onConnectionCompleted(ReaderDevice reader, Throwable err
```

Activate scanning

```
readerDevice.startScanning();
```

You can stop scanning with the following:

```
readerDevice.stopScanning();
```

The onReadResultReceived listener method (see Step 3) will be invoked as a barcode was decoded by the

reader, or the scanning process has finished.

Configuring the Reader Device

After connecting to the scanning device, you may want (or need) to change some of its settings. The cmbSDK provides a set of high-level, device independent APIs for setting and retrieving the current configuration of the device.

Like in the case of initializing the Reader Device, there are some differences between using an MX reader and the camera reader for scanning. These differences are detailed in the following sections.

MX Mobile Terminals

The MX family of mobile terminals provides sophisticated device configuration and management, including saved configurations on the device itself. In general, these devices come from Cognex preconfigured for an exceptional out-of- the-box experience with most symbologies and features ready to use.

When custom reconfiguration is desired, this is typically done using either the DataMan Setup Tool, or the DataMan Quick Setup as these tools can be used to distribute saved configurations easily to multiple devices, thereby greatly simplifying configuration management.

However, it is still possible (and sometimes desirable) for the mobile application itself to configure the MX device:

- You can have multiple scanning applications, each of which requires a different set of device settings.
- You may simply want to guarantee that the certain options are in a “known” state and not rely on the fact that the device has been preconfigured correctly.

Built-in Camera

Much like an MX mobile terminal, the cmbSDK employs a default set of options for barcode reading with the built-in camera of the mobile device, providing a good out-of-box experience. However, there are two important differences to keep in mind:

- The cmbSDK does not implement saved configurations for the camera reader. This means that every time an application that uses the camera reader starts, it starts with its defaults.
- The cmbSDK does not enable any *symbologies* by default: you as the application programmer must enable all barcode symbologies your application needs to scan. By requiring the application program to explicitly enable only the symbologies it needs, the most optimal scanning performance can be achieved. This concept was more thoroughly discussed in the Overview section.

Enabling Symbologies

Individual symbologies can be enabled using the following method of the Reader Device object:

```
public void setSymbologyEnabled(final Symbology symbology, final boolean enable, final OnSymbologyListener listener)
readerDevice.setSymbologyEnabled(Symbology.DATAMATRIX, true, null);
readerDevice.setSymbologyEnabled(Symbology.UPC_EAN, true, null);
```

All symbologies used for the symbology parameter in this method can be found in **ReaderDevice.java**.

Examples

```
 /* Enable QR scanning */
readerDevice.setSymbologyEnabled(Symbology.QR, true, null);
```

The same method can also be used to turn symbologies off:

```
/ * Disable Code 25 scanning */ readerDevice.setSymbologyEnabled(Symbology.C25, false, null);
```

You can implement the method for OnSymbologiesListener:

```
@Override
public void onSymbologyEnabled(ReaderDevice reader, Symbology symbology, Boolean enabled, Throwable error) {
if (error != null) {
/* Unsuccessful
probably the symbology is unsupported by
the current device, or there is a problem with the connection between
the readerDevice and MX device
*/
} else {
// Success }
}
```

Resetting the Configuration

The cmbSDK includes a method for resetting the device to its default settings. In the case of an MX mobile terminal, this is the configuration saved by default (not the factory defaults), while in the case of the built-in camera, these are the defaults identified in Appendix B, where no symbologies will be enabled. This method is the following:

```
readerDevice.resetConfig(null);
```

Being an async method, you can monitor its completion using OnResetConfigListener interface passed as an optional parameter to the method.

```
public class ScannerActivity extends Activity implements .... OnResetConfigListener .... { ....
@Override
public void onResetConfigCompleted(ReaderDevice reader, Throwable error) {
if (error != null) { // Unsuccessful
} else {
// Success }
}
```

Illumination Control

If your reader device is equipped with illumination lights (e.g. LEDs), you can control whether they are ON or OFF when scanning starts using the following method of your Reader Device object:

```
readerDevice.setLightsOn(true, null);
```

Optionally, you can implement the interface method for OnLightsListener, which is the second parameter of the method.

```
public class ScannerActivity extends Activity implements .... OnLightsListener .... { ....
@Override
public void onLightsOnCompleted(ReaderDevice reader, Boolean on, Throwable error) {
if (error != null) { // Unsuccessful
} else {
// Success }
} }
```

Keep in mind that not all devices and device modes supported by the cmbSDK allow for illumination control. For example, if using the built-in camera in passive aimer mode, illumination is not available since the LED is being used for aiming.

Advanced Configuration

Every Cognex scanning device implements DataMan Control Commands (DMCC), a method for configuring and controlling the device. Virtually every feature of the device can be controlled using this text based language. The API provides a method for sending DMCC commands to the device. Commands exist both for setting and querying configuration properties.

Appendix A includes the complete DMCC reference for use with the camera reader. DMCC commands for other supported devices (e.g. the MX-1000) are included with the documentation of that particular device.
Appendix B provides the default values for the camera reader's configuration settings as related to the corresponding DMCC setting.
The following examples show different DMCC commands being sent to the device for more advanced configuration.

Examples

```
//Change the scan direction to omnidirectional
readerDevice.getDataManSystem().sendCommand("SET DECODER.1D-SYMBOLORIENTATION 0", ScannerActivity.this);
//Change live-stream preview's scanning timeout to 10 seconds
readerDevice.getDataManSystem().sendCommand("SET DECODER.MAX-SCAN-TIMEOUT 10", ScannerActivity.this);
```

You can also invoke DMCC query commands and receive their response in the *OnResponseReceivedListener.onResponseReceived()* method.

```
//Get the type of device connected readerDevice.getDataManSystem().sendCommand("GET DEVICE.NAME", new OnResponseReceivedLis
@Override
public void onResponseReceived(DataManSystem dataManSystem, DmccResponse dmccResponse) {
if (dmccResponse.getError() != null) {
// Unsuccessful
Log.e("DMCC_ERR", “GET DEVICE.NAME failed”,dmccResponse.getError());
} else {
// Success - Use the following result fields:
//int mResponseId = dmccResponse.getResponseId(); //String mPayLoad = dmccResponse.getPayLoad(); //byte[] mBinaryData = dmc
} );
}
```

Scanning Barcodes

With a properly configured reader, you are now ready to scan barcodes. This can be done by calling the startScanning method from your Reader Device object.

What happens next is based on the type of Reader Device and how it has been configured, but in general:

- If using an MXreader, the user can now press a trigger button on the device to turn the scanner on and read a barcode.
- If using the camera reader, the cmbSDK starts the camera, displays the configured live-stream preview, and begins analyzing the frames from the video stream, looking for a configured barcode symbology.

Scanning stops under one of the following conditions:

- The reader found and decoded a barcode.
- The user released the trigger or pressed the stop button on the live-stream preview screen.
- The camera reader timed out with out finding a barcode.
- The application itself calls the *stopScanning()* method.

When a barcode is decoded successfully (the first case), you will receive a ReadResults iterable result collection object in ReaderDevice listener method, already implemented in Step 3.

```
@Override
public void onReadResultReceived(ReaderDevice reader, ReadResults results) {
 if (results.getCount() > 0) {
  ReadResult result = results.getResultAt(0);

  // USE String symbologyName; String code; Bitmap frame; VARIABLES IN YOUR APPLICATION

  if (result.isGoodRead()) {
   String symbologyName;
   String code = result.getReadString();
   Symbology symbology = result.getSymbology();

   if (symbology != null) {
    symbologyName = symbology.getName();
    tvSymbology.setText(symbologyName);
   } else {
    tvSymbology.setText("UNKNOWN SYMBOLOGY");
   }

   tvCode.setText(code);

  } else {
   tvSymbology.setText("NO READ");
   tvCode.setText("");
  }

  Bitmap frame = result.getImage();
  ivPreview.setImageBitmap(frame);
 }
// READY TO SCAN AGAIN
}
```

In the above example, ivPreview is an ImageView used to display an image of the barcode that was scanned, and tvCode is a TextView used to show the result from the barcode. You can also use the BOOL from result.isGoodRead() to check whether the scan was successful or not.

Working with Results

When a barcode is successfully read, a ReadResult object is created and returned by the onReadResultReceived method. In case of having multiple barcodes successfully read on a single image/frame, multiple ReadResult objects are returned in the ReadResult object.

The ReadResult class has properties describing the result of a barcode read:

- **is GoodRead()** (boolean) : tells whether the read was successful or not
- **get ReadString()** (String): the decoded barcode as a string
- **get Image()** (Bitmap): the image/frame that the decoder has processed
- **get ImageGraphics()** (String): the boundary path of the barcode as SVG data
- **get Xml()** (String): the raw XML that the decoder returned
- **get Symbology** (Symbology): the symbology type of the barcode.This enum is defined in ***ReaderDevice.java.***

When a scanning ends with no successful read, a *ReadResult* is returned with the *goodRead* property set to false. This usually happens when scanning is canceled or timed out.

To enable the image and imageGraphics properties being filled in the *ReadResult* object, you have to set the corresponding *enableImage()* and/or *enableImageGraphics()* properties of the ReaderDevice object.

To see an example on how the image and SVG graphics are used and displayed in parallel, refer to the sample applications provided in the SDK package.

```
Not all supported devices provide SVG graphics.
```

To access the raw bytes from the scanned barcode, you can use the XML property. The bytes are stored as a Base64 String under the "full_string" tag. Here's an example how you can use a XML parser to extract the raw bytes from the XML property.

```
try {
    XmlPullParserFactory factory = XmlPullParserFactory.newInstance();
    factory.setNamespaceAware(true);
    XmlPullParser xpp = factory.newPullParser();

    String tag = "";

    // the raw bytes will be stored in this variable
    byte[] bytes;

    xpp.setInput(new StringReader(result.getXml()));
    int eventType = xpp.getEventType();
    while (eventType != XmlPullParser.END_DOCUMENT) {
        if (eventType == XmlPullParser.START_TAG) {
            tag = xpp.getName();
        }
        else if (eventType == XmlPullParser.TEXT && tag.equals("full_string")) {
            String base64String = xpp.getText();
            // Get the bytes from the base64 string here
            bytes = Base64.decode(base64String, Base64.DEFAULT);
            break;
        }
        else if (eventType == XmlPullParser.END_TAG && tag.equals("full_string")) {
            tag = "";
            break;
        }
        eventType = xpp.next();
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

Image Results

By default, the image and SVG results are disabled, which means that when scanning, the ReadResults will not contain any data in the corresponding properties.

To enable image results, invoke the enableImage() method from the ReaderDevice object:

```
readerDevice.enableImage(true);
```

To enable SVG results, invoke the enableImageGraphics() method on ReaderDevice object:

```
readerDevice. enableImageGraphics(true);
```

Handling Disconnects

There might be cases when a device disconnects due to low battery condition or manual cable disconnection. These cases can be detected by the onConnectionStateChanged() method of the ReaderDeviceListener interface.

Note: The onAvailabilityChanged() method of ReaderDeviceListener is also called when the device becomes physically unavailable. It means that the (re)connection is not possible. You should always check the getAvailability() method of the ReaderDevice object before trying to call the connect() method.