

# **ZEBRA SCANNER SDK for ANDROID DEVELOPER GUIDE**



# **ZEBRA SCANNER SDK for ANDROID DEVELOPER GUIDE**

MN002223A03

Revision A

September 2016

No part of this publication may be reproduced or used in any form, or by any electrical or mechanical means, without permission in writing from Zebra. This includes electronic or mechanical means, such as photocopying, recording, or information storage and retrieval systems. The material in this manual is subject to change without notice.

The software is provided strictly on an “as is” basis. All software, including firmware, furnished to the user is on a licensed basis. Zebra grants to the user a non-transferable and non-exclusive license to use each software or firmware program delivered hereunder (licensed program). Except as noted below, such license may not be assigned, sublicensed, or otherwise transferred by the user without prior written consent of Zebra. No right to copy a licensed program in whole or in part is granted, except as permitted under copyright law. The user shall not modify, merge, or incorporate any form or portion of a licensed program with other program material, create a derivative work from a licensed program, or use a licensed program in a network without written permission from Zebra. The user agrees to maintain Zebra’s copyright notice on the licensed programs delivered hereunder, and to include the same on any authorized copies it makes, in whole or in part. The user agrees not to decompile, disassemble, decode, or reverse engineer any licensed program delivered to the user or any portion thereof.

Zebra reserves the right to make changes to any software or product to improve reliability, function, or design. Zebra does not assume any product liability arising out of, or in connection with, the application or use of any product, circuit, or application described herein.

No license is granted, either expressly or by implication, estoppel, or otherwise under any Zebra Technologies Corporation, intellectual property rights. An implied license only exists for equipment, circuits, and subsystems contained in Zebra products.

---

## Warranty

For the complete Zebra hardware product warranty statement, go to:

<http://www.zebra.com/warranty>.

---

## Revision History

Changes to the original manual are listed below:

Change	Date	Description
-01 Rev A	8/2015	Initial Release
-02 Rev A	10/2015	Software updates.
-03 Rev A	9/2016	Software updates.

# TABLE OF CONTENTS

Warranty .....	ii
Revision History .....	ii

## About This Guide

Introduction .....	v
Chapter Descriptions .....	v
Related Documents .....	v
Additional Resources .....	vi
Notational Conventions .....	vi
Service Information .....	vi

## Chapter 1: GETTING STARTED with the ZEBRA SCANNER SDK for ANDROID

Introduction .....	1-1
Overview of the Zebra Scanner SDK for Android .....	1-1
Supported Scanners .....	1-2
System Requirements .....	1-3
Installation and Configuration .....	1-3
Installing the Scanner Control Application .....	1-3
Running and Configuring the Scanner Control Application .....	1-4
Using Scanner Control Application with a Supported Device .....	1-7
Setting Up the Zebra Scanner SDK for Android in Android Studio .....	1-15
Prerequisite 1 - Installation of Android Studio .....	1-15
Prerequisite 2 - Configuring the Host to Communicate With the Device .....	1-15
Installing and Building the Android SDK Project .....	1-15

## Chapter 2: ANDROID DEVELOPMENT SDK

Introduction .....	2-1
Initialization .....	2-1
SDK Initialization .....	2-1
Setting SDK Handler Delegate .....	2-2
Setting Operation Mode .....	2-2

Subscribing to Events .....	2-3
Connecting to a Scanner .....	2-4
Detecting Available Scanners .....	2-4
Connecting to an Available Scanner .....	2-5
Receiving Bar Code Data .....	2-7
Retrieving Scanner Attributes .....	2-8
Sending Remote Commands .....	2-12
Beep the Beeper .....	2-12
Disabling a Bar Code Symbology Type .....	2-13
Disabling the Scanner .....	2-14

# ABOUT THIS GUIDE

---

## Introduction

The *Zebra Scanner SDK for Android Developer Guide* provides installation and programming information for the Software Developer Kit (SDK) that allows Software Decode based applications for Android based devices.

---

## Chapter Descriptions

This guide includes the following topics:

- [Chapter 1, GETTING STARTED with the ZEBRA SCANNER SDK for ANDROID](#) provides information about the Android Software Development Kit (Android SDK).
- [Chapter 2, ANDROID DEVELOPMENT SDK](#) describes how to connect to a scanner through the SDK, retrieve bar codes, and send command and control messages using the included application as an example.

---

## Related Documents

- *RFD8500 Developer Guide*, p/n MN002222Axx.
- *Zebra Scanner SDK for iOS Developer Guide*, p/n MN001834Axx.
- *RFD8500 User Guide*, p/n MN002065Axx.
- *RFD8500 Quick Start Guide*, p/n MN002225Axx.
- *RFD8500 Regulatory Guide*, p/n MN002062Axx.
- *CRD1S-RFD8500 (1-Slot), CRDUNIV-RFD8500-1R (3-Slot), CRD4S-RFD8500 (4-Slot) Universal Charge Only Cradles Regulatory Guide*, p/n MN002224Axx.
- *DS3678 Cordless Digital Imager Product Reference Guide*, p/n MN-002689-xx.
- *CS4070 Product Reference Guide*, p/n MN000762Axx.

For the latest version of this guide and all guides, go to: [www.zebra.com/support](http://www.zebra.com/support).

---

## Additional Resources

For further information on the various topics covered in this *Developer Guide*, also refer to:

- *Android Studio Overview* at <http://developer.android.com/tools/studio/index.html>
- *Android Studio Training and Samples* at <https://developer.android.com/training/index.html>
- *Android API Guides* at <http://developer.android.com/guide/index.html>

---

## Notational Conventions

This document uses the following conventions:

- *Italics* are used to highlight chapters, sections, field names, and screen names in this and related documents.
- Courier New font type is used to represent code snippets.
- bullets (•) indicate:
  - Action items
  - Lists of alternatives
  - Lists of required steps that are not necessarily sequential
- Sequential lists (e.g., those that describe step-by-step procedures) appear as numbered lists.



**NOTE** This symbol indicates something of special interest or importance to the reader. Failure to read the note does not result in physical harm to the reader, equipment or data.



**CAUTION** This symbol indicates that if this information is ignored, the possibility of data or material damage may occur.



**WARNING!** This symbol indicates that if this information is ignored the possibility that serious personal injury may occur.

---

## Service Information

If you have a problem using the equipment, contact your facility's technical or systems support. If there is a problem with the equipment, they contact the Zebra Technologies Global Customer Support Center at: <http://www.zebra.com/support>.

When contacting Zebra support, please have the following information available:

- Product name
- Version number

Zebra responds to calls by e-mail, telephone or fax within the time limits set forth in support agreements.

If your problem cannot be solved by Zebra support, you may need to return your equipment for servicing and will be given specific directions. Zebra is not responsible for any damages incurred during shipment if the approved shipping container is not used. Shipping the units improperly can possibly void the warranty.

If you purchased your business product from a Zebra business partner, contact that business partner for support.



# Chapter 1 GETTING STARTED with the ZEBRA SCANNER SDK for ANDROID

---

## Introduction

This chapter provides information about the Zebra Scanner SDK for Android - an architectural framework providing a single programming interface to provide two way communication between Android based applications and supported Zebra scanning devices.

---

## Overview of the Zebra Scanner SDK for Android

The Android SDK provides an abstraction layer, delivered in the form of an Android Library (.aar file). This library provides an API to provide all of the connection, command and control, and communication facilities required to operate a Zebra scanner on the Android platform.

The Android SDK is broken into three parts:

- **Scanner Control Application for Android** - Installable application for Android devices to enable quick testing and demonstration of the SDK capabilities on a Bluetooth® (BT) supported Zebra scanner or a USB SNAP! scanner. The Scanner Control application is available for download from the Google Play Store and is distributed within the SDK.
- **Android SDK Library File** - This is provided as standalone AAR file that can be imported into a new Android Application. Android Studio is the development environment that is recommended.
- **Android SDK Demo Application Project** - This is a zip file containing the Scanner Control application project files from Android Studio. It contains the source code to the application and the SDK library files necessary to build, test and modify the application as necessary.

## Supported Scanners

Currently the following Zebra scanners are supported:

### USB SNAP I

- PL3307
- DS457
- DS4308
- LS2208
- DS6878 and Presentation Cradle
- MP6210 (CSS + Scale) + EAS (Sensormatic).

### Bluetooth

- CS4070 (in Bluetooth SSI Profile mode)
- LI4278 (in SSI Host Server mode or Cradle Host mode by scanning pairing bar code)
- DS6878 (in SSI Host Server mode or Cradle Host mode by scanning pairing bar code)
- RFD8500 (in default mode)
- DS3678 (In SSI BT Classic mode)
- LI3678 (In SSI BT Classic mode).



**NOTE** To configure a device in the mode specified, refer to the appropriate Product Reference Guide, User Guide, or Integration Guide.

[Table 1-1](#) lists the pairing methods that can be used for each scanner model.

**Table 1-1** *Pairing Methods*

Scanner Model	Pairing Bar Code		Manual Pairing
	Legacy	ScanToConnect	
CS4070	Yes		Yes
DS3678	Yes	Yes	Yes
LI3678	Yes	Yes	Yes
DS6878	Yes		Yes
LI4278	Yes		Yes
RFD8500			Yes

## System Requirements

The following system requirements are necessary to use the Scanner Control application (demo application).

- Hardware device supporting Android KitKat version 4.4, or later, with Bluetooth and USB port (only if using USB scanners).

✓ **NOTE** Android version Marshmallow (6.x) is the latest version that was tested.

The following system requirements are necessary to develop and test applications, and to use the Scanner Control application project for development and testing.

- Android Studio 1.3 or later installed on Windows or Linux
- Android API Level 19 or later (Demo application was built and tested with API Level 19)
- The Scanner Control Application Project (packaged as an Android Studio Project)
- Hardware device running Android Kit Kat 4.4 or later, or emulator. Note that in order to create a connection to the scanner, a Bluetooth connection is required.

---

## Installation and Configuration

### Installing the Scanner Control Application

The Scanner Control application can be installed directly onto a mobile device. The following steps include general guidelines for installation. Menus and options may vary depending on the version of Android running.

✓ **NOTE** The Scanner Control application is available for download from the Google Play Store and is distributed within the SDK.

To install the demo application:

1. Previous versions (earlier than v1.0.16.0 of the Zebra Scanner Control application used a different name and branding signature and must be manually uninstalled before installing the current version of the SDK demo application.


To do this:

- a. Go to *Android Settings* > *Application Manager* (this varies depending on the version and platform of Android running).
  - b. Select the *Android Scanner Demo App*.
  - c. Select *Clear Cache* and *Clear Data* to remove any resident demo settings.
  - d. Select *Uninstall* to remove the demo application from the system.
2. Install the application using one of the following methods:
    - a. Using Google Play Store:
      - i. Go to <https://play.google.com/store/apps/details?id=com.zebra.scannercontrol.app> or search for Scanner Control in Google Play Store.
      - ii. Install the Scanner Control application.

or

    - b. Manual Installation from the SDK package:

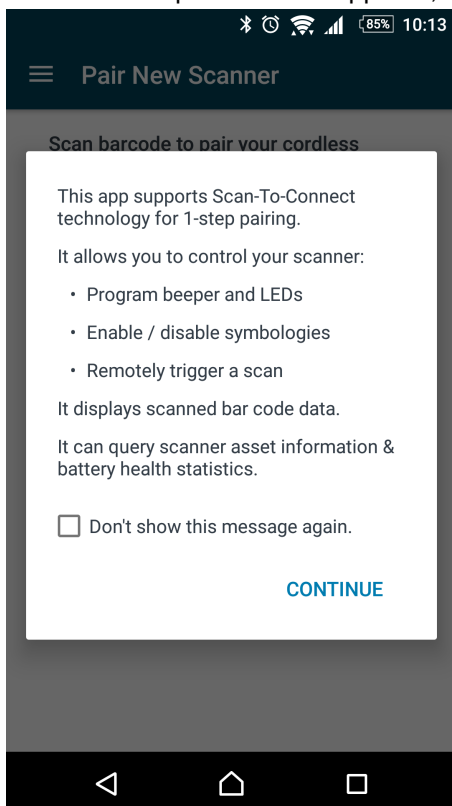
- i. Copy the file `scanner_control_app_version.apk` file included with the SDK package to the Android device.
- ii. Navigate to the saved location and select the APK package file.
- iii. The Android OS provides a warning that the application is from an untrusted source and requires that installation from unknown sources be enabled for this installation. This is normal. Select the option to install from unknown sources.

3. The Android OS installs the application and installs a *Scanner Control Application* icon (  ) in the *Apps* menu.

## Running and Configuring the Scanner Control Application

To run the application:

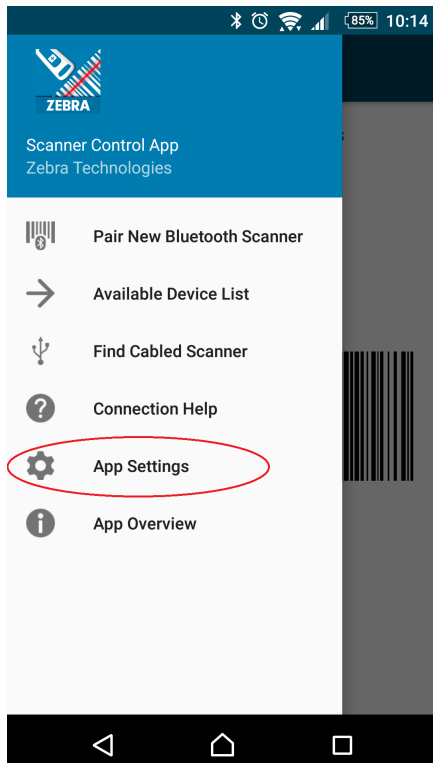
1. Select the Scanner Control application from the **Android App** menu.
2. After the splash screen appears, the overview message screen displays (*Figure 1-1*).



**Figure 1-1** *Scanner Control App - Overview*

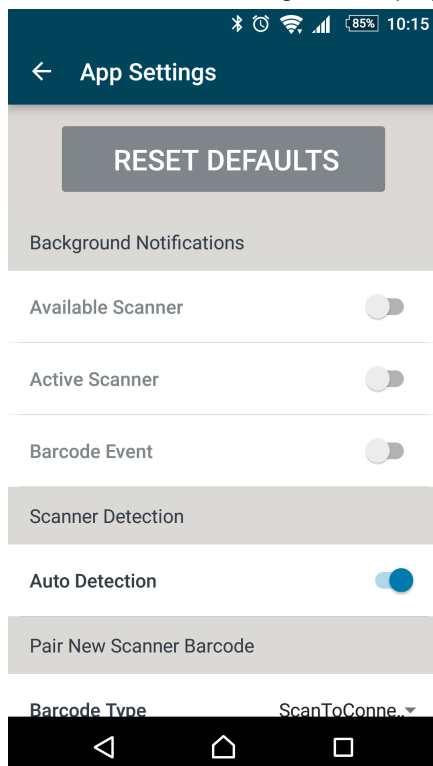
3. Select **CONTINUE**. The **Pair New Scanner** bar code displays.

4. From the **Scanner Control App**, select **App Settings** to configure the application.



**Figure 1-2** Scanner Control App Menu - App Settings

5. The available settings are displayed in [Figure 1-3](#). See [Table 1-2](#) for settings descriptions.



**Figure 1-3** App Settings

**Table 1-2** *App Settings Descriptions*

<b>Setting</b>	<b>Description</b>
Reset Defaults	Changes all settings to the defaults.
<b>Background Notifications</b>	
Available Scanner	Notifies the user when a new scanner is available for connection.
Active Scanner	Notifies the user when a scanner becomes connected.
Barcode Event	Notifies the user when a bar code scans.
<b>Scanner Detection</b>	
Auto Detection	Automatically detects new scanners when they are paired or connected via USB.
<b>Pair New Scanner Barcode</b>	
Barcode Type	<p>Pairing bar code type to be displayed on the home screen of the application:</p> <ul style="list-style-type: none"> <li>• Legacy Pairing Scanner must be configured in the correct host mode to be paired and connected.</li> <li>• ScanToConnect Suite Scanner automatically changes its communication protocol and configuration (set factory defaults or keep current configuration) as configured in the Communication Protocol and Set Factory Defaults settings before connecting.</li> </ul>
Communication Protocol	Communication protocol to be used in ScanToConnect Suite bar code. Application can connect only in the SSI over Bluetooth Classic protocol.
Set Factory Defaults	Determines whether or not the scanner changes its settings to factory defaults before connecting.

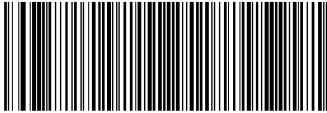
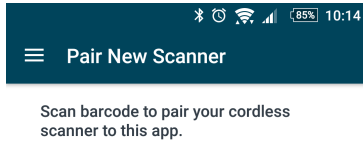
## Using Scanner Control Application with a Supported Device



**IMPORTANT** This application demonstrates the various capabilities of the lower level SDK library, and is not intended to be used for production functions.

To run the application:

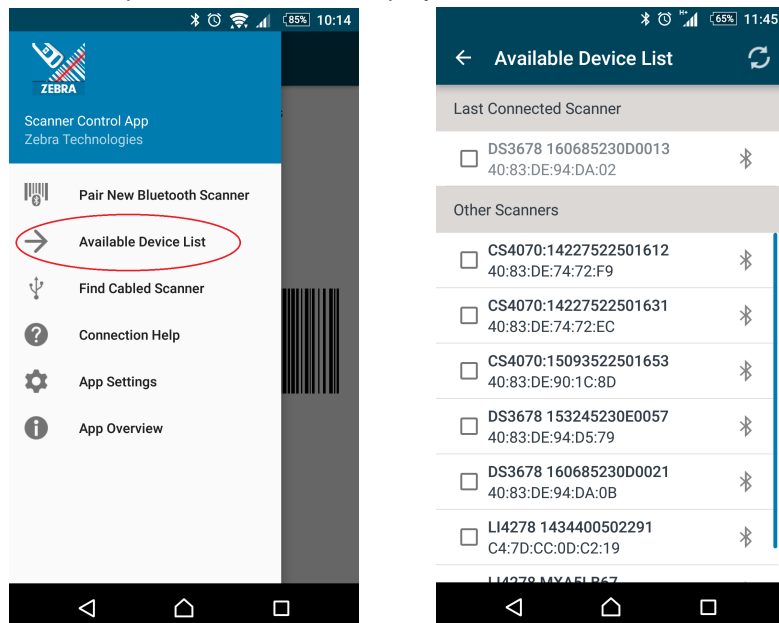
1. There are two ways to connect a Bluetooth scanner.
  - a. Scan the pairing bar code. When the scanner and pairing bar code are configured correctly, scan the pairing bar code to connect.



**Figure 1-4** Pairing Bar Code

or

- b. Pair the scanner manually. To pair the scanner manually, refer to the appropriate Product Reference Guide, User Guide, or Integration Guide for instructions on how to pair the specific device.
2. When paired the scanner displays in the Available Device List.



**Figure 1-5** Available Device List

3. If using a USB scanner:
  - a. Configure its USB host mode to SNAPI.
  - b. Connect it to the Android device
  - c. The scanner appears in the Available Device List.or
  - d. Select **Find Cabled Scanner** from the menu to display the SNAPI bar code.

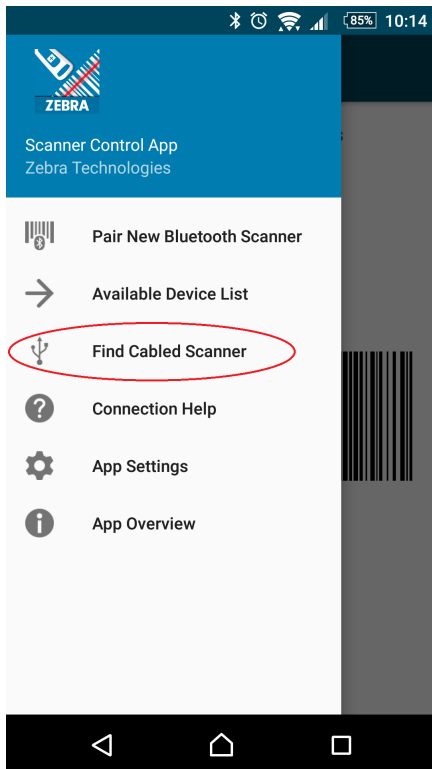
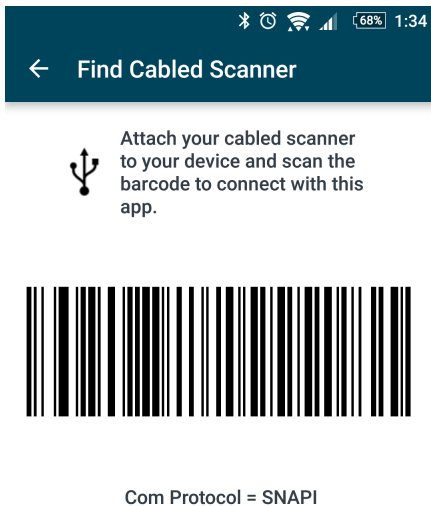


Figure 1-6 Menu - Find Cabled Scanner



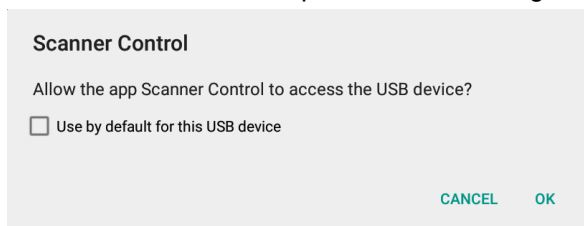
- e. If there is a single scanner to connect, the application connects to the scanner automatically. If there are no SNAPI scanners connected, the application displays the SNAPI bar code ([Figure 1-7](#)) to scan to connect.



**Figure 1-7** SNAPI Bar Code

- f. If multiple USB SNAPI scanners are available, the application displays the available scanner list from which the user can select the appropriate scanner to which the application should connect.

It may be necessary for the user to give the Android operating system permission to access the USB device. Should a permissions message display, select OK.



**Figure 1-8** Application Permission

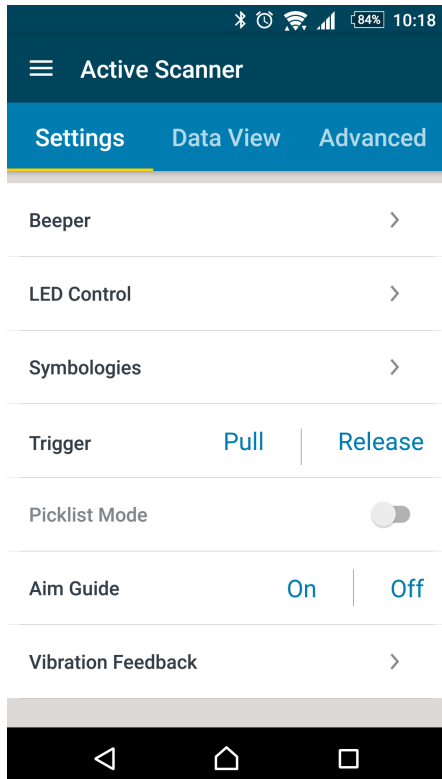
4. When a device is selected in the [Available Device List on page 1-7](#), the application attempts to connect to the scanner. When the connection is made, [Figure 1-9](#) displays.



**NOTE** Scanning the pairing bar code or clicking **Find Cabled Scanner** displays this screen.

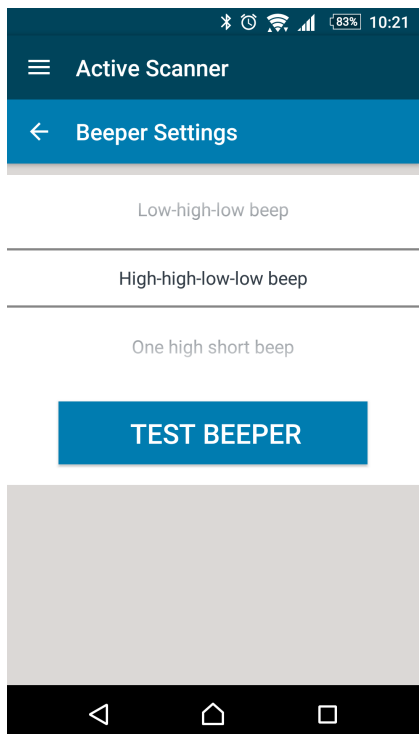
## Settings

From this screen you can exercise various options.



**Figure 1-9** Settings - Active Scanner (Connected Device)

## Beeper



**Figure 1-10** Beeper Settings

**LED**

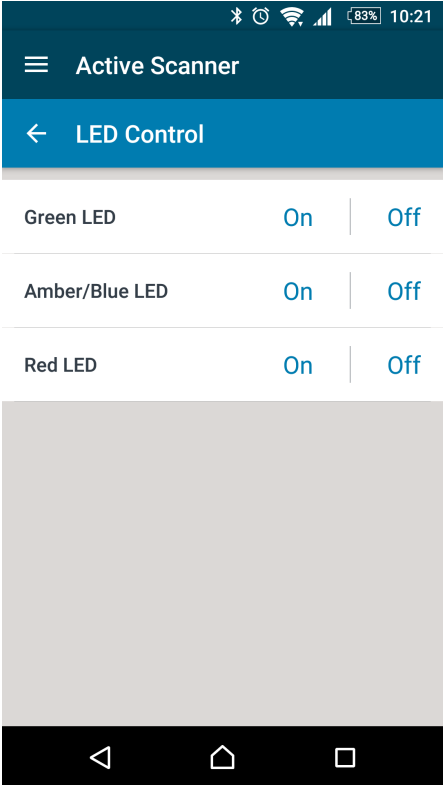


Figure 1-11 LED Control

**Symbologies**

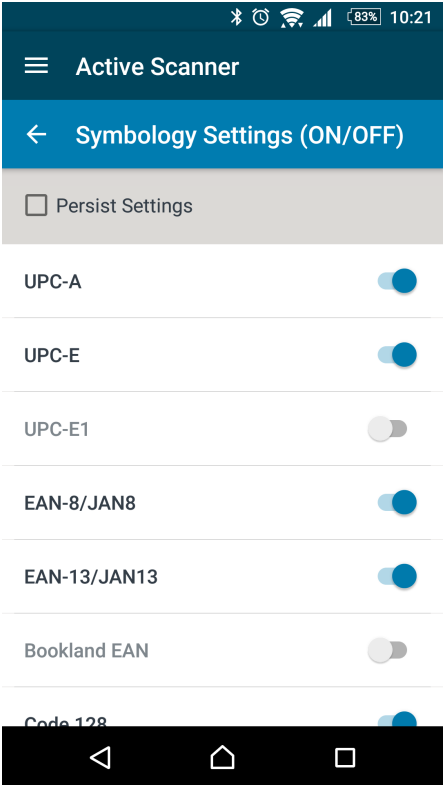


Figure 1-12 Symbology Settings

### Vibration Feedback

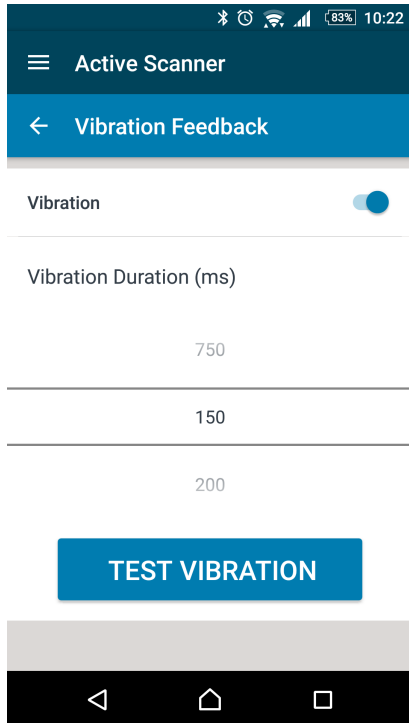


Figure 1-13 Vibration Feedback

### Data View

Select the Data View tab to see bar codes received.

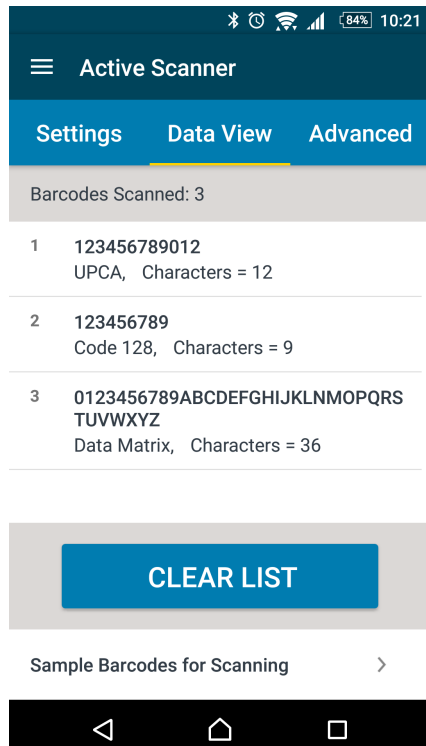


Figure 1-14 Data View Tab

**Advanced**

Select the **Advanced** tab to **Find Scanner** and select an option to display [Asset Information on page 1-13](#), [Battery Statistics on page 1-14](#) and to [Update Firmware on page 1-14](#).

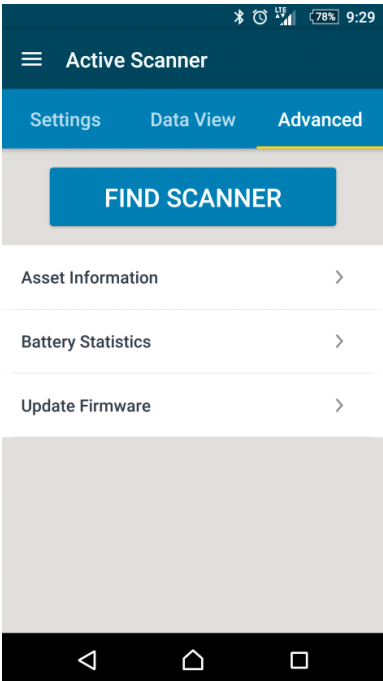


Figure 1-15 Advance Tab - Find Scanner

**Asset Information**

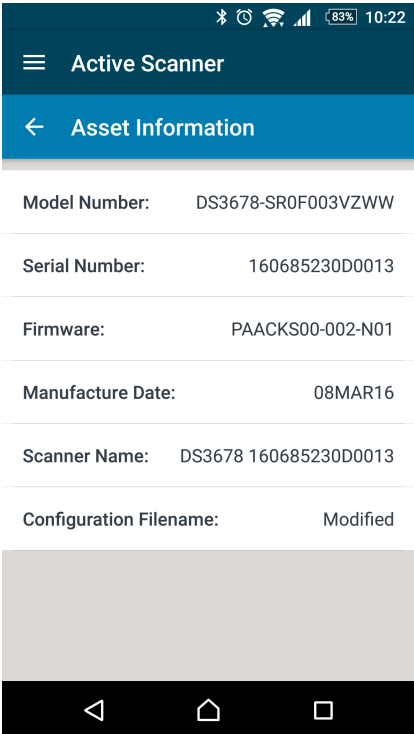


Figure 1-16 Advance Tab - Asset Information

**Battery Statistics**

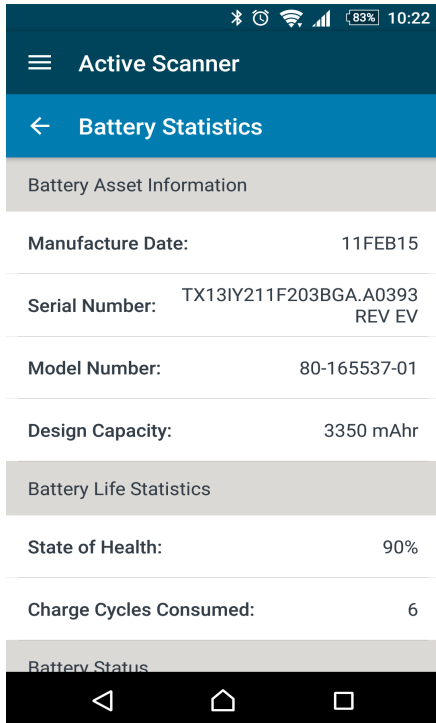


Figure 1-17 Advanced Tab - Battery Statistics

**Update Firmware**

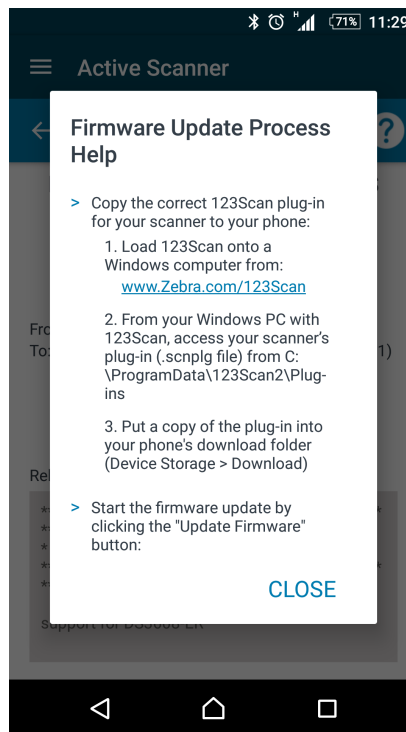


Figure 1-18 Advanced Tab - Update Firmware and Firmware Update Process Help

---

## Setting Up the Zebra Scanner SDK for Android in Android Studio

### Prerequisite 1 - Installation of Android Studio

To install Android Studio:

1. Download Android Studio from the Android developers site: <https://developer.android.com/sdk/index.html>.
2. Before running the Android Studio, run the SDK Manager to install API Level 19. For additional instructions, go to: <http://developer.android.com/tools/help/sdk-manager.html>.
3. The Android Studio uses the Gradle build system. For more information and an overview of the Gradle build system, review the documentation at: <https://developer.android.com/sdk/installing/studio-build.html>.

### Prerequisite 2 - Configuring the Host to Communicate With the Device

Some devices require that you download and install a driver so that is recognized as a USB device on Windows. If using Linux, the device has to be configured manually. Review the documentation regarding installing the appropriate driver for the device: <http://developer.android.com/tools/device.html>.

### Installing and Building the Android SDK Project

To install and build the Android SDK project:

1. Unzip the file *android\_scanner\_sdk\_demo\_version\_src.zip* into a local directory. This directory is referred to as *dev\_directory*.
2. Open Android Studio and select *Open an Existing Project*.
3. Navigate to the *dev\_directory* and select the *AndroidScannerSDK* folder:  
`dev_directory\AndroidScannerSDK`  
As this is the first time that Android Studio is opening the project, it begins creating all of the necessary local project files and indexes. This may take a minute or two.
4. Copy the folder to the development host. This directory is called *dev\_directory*.
5. Open the Android Studio.
6. Select *Open an Existing Android Studio Project*.
7. Navigate to the *dev\_directory* and select the project *AndroidScannerSDK* under the *src* folder.
8. The *Project* tab on the left panel displays the contents of the *Project*, including the *app* project which is the top level for the application source.
9. Select the *Build* option from the *Menu* bar. Click **Make Project**. This builds the APK application located under: `dev_directory\AndroidScannerSDK\app\build\outputs\apk`.





# Chapter 2 ANDROID DEVELOPMENT SDK

---

## Introduction

This chapter outlines the steps to connect to a scanner through the SDK, retrieve bar codes, and send commands and control messages using the included Scanner Control application as an example. Relevant code snippets are included.

---

## Initialization

### SDK Initialization

Before the application can use the underlying SDK, it must first be initialized and set up to communicate with it. The `Application` class defined in the application, which extends the required `android.app.Application` class creates the `SDKHandler` object. The following application code shows how this is done.

#### Code Snippet - SDK Initialization Code

```
public class Application extends android.app.Application {  
  
    //Instance of SDK Handler  
    public static SDKHandler sdkHandler;  
  
    ...  
    ...  
  
    //Barcode data  
    @Override  
    public void onCreate() {  
        super.onCreate();  
        sdkHandler = new SDKHandler(this);  
    }  
  
    ...  
    ...  
}
```

When this object is passed to the `SDKHandler`, all necessary library initialization is performed and the reference is stored as an Android Context object which is defined by Android 4.4+ as:

An "Interface to global information about an application environment. This is an abstract class whose implementation is provided by the Android system. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting and receiving intents, etc."

The `Application` object can now provide the Interface to all SDK API methods. These are methods that call "into" the SDK. Methods that call "out" (callbacks) are defined by the SDK API Delegate as shown in [Setting SDK Handler Delegate on page 2-2](#).

## Setting SDK Handler Delegate

In order for the parent class to receive events from the SDK as well as call SDK methods, it must define a delegate that conforms to the `IDcsSdkApiDelegate` interface definition.

### Code Snippet - Setting SDK API Delegate

```
public class BaseActivity extends ActionBarActivity implements ScannerAppEngine,
IDcsSdkApiDelegate {

    // The Handler that gets information back from the BluetoothChatService
    protected final Handler mHandler = initializeHandler();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mScannerInfoList=Application.mScannerInfoList;
        TAG = getClass().getSimpleName();

        // Setting up the SDK delegate to receive events
        Application.sdkHandler.dcssdkSetDelegate(this);
        initializeDcssdkWithAppSettings();
    }
}
```

The code that passes the `BaseActivity` as the defined SDK delegate, as shown in [Code Snippet - Setting SDK API Delegate](#), now contains all of the necessary inherited methods to receive events from the SDK to determine scanner connection status, scanner availability, and the various data events that are sent up from the SDK. Note that this has to be set explicitly because the `Application` class owns the `sdkHandler` object. As stated earlier, the global `Application` class is used to access the SDK API as shown in this example.

## Setting Operation Mode

A client can set the operation mode interested. Currently Bluetooth and SNAPI are supported. You can set multiple operation modes.

### Code Snippet - Setting Operation Mode

```
Application.sdkHandler.dcssdkSetOperationalMode(DCSSDKDefs.DCSSDK_MODE.DCSSDK_OPMODE_BT_NO
RMAL);

Application.sdkHandler.dcssdkSetOperationalMode(DCSSDKDefs.DCSSDK_MODE.DCSSDK_OPMODE_SNAPI
);
```

## Subscribing to Events

A client can subscribe to specific event types:

- `DCSSDK_EVENT_BARCODE` - Notifies the client of an available bar code from the active scanner.
- `DCSSDK_EVENT_SCANNER_APPEARANCE` - Notifies the client that a scanner that was paired to the host is available for connection.
- `DCSSDK_EVENT_SCANNER_DISAPPEARANCE` - Notifies the client that a scanner is no longer available for connection.
- `DCSSDK_EVENT_SESSION_ESTABLISHMENT` - Notifies the client that a connection to a scanner has become active (connected and available to communicate).
- `DCSSDK_EVENT_SESSION_TERMINATION` - Notifies the client that a connection to a scanner has terminated (can no longer be communicated with).

### Code Snippet - Subscribing to Events

Following is an example of how to subscribe to these events:

```
int notifications_mask = 0;

// We would like to subscribe to all scanner available/not-available events
notifications_mask |=
    DCSSDKDefs.DCSSDK_EVENT.DCSSDK_EVENT_SCANNER_APPEARANCE.value |
    DCSSDKDefs.DCSSDK_EVENT.DCSSDK_EVENT_SCANNER_DISAPPEARANCE.value;

// We would like to subscribe to all scanner connection events
notifications_mask |=
    DCSSDKDefs.DCSSDK_EVENT.DCSSDK_EVENT_SESSION_ESTABLISHMENT.value |
    DCSSDKDefs.DCSSDK_EVENT.DCSSDK_EVENT_SESSION_TERMINATION.value;

// We would like to subscribe to all barcode events
notifications_mask |= DCSSDKDefs.DCSSDK_EVENT.DCSSDK_EVENT_BARCODE.value;

// subscribe to events set in notification mask
Application.sdkHandler.dcssdkSubscribeForEvents(notifications_mask);

// ...
```

In order for the client application to receive these events, the application must have a defined class that implements the `IDcsSdkApiDelegate` interface. The application accomplishes this through the `BaseActivity` class. Since all `Activity` classes extend the `BaseActivity`, all `UI Activity` classes can receive these events.

---

## Connecting to a Scanner

This section describes how to perform the initial setup, discovery and connection to a scanner device via Bluetooth using the Scanner Control application for Android as a reference example. For each topic, a link to the Android Studio developer site is included.

### Detecting Available Scanners

When scanners are paired to the Android host, they are provided to the application. The SDK must be told to look for these scanners (some use cases would disable this feature). At the same time, any events the client application is interested in should be subscribed to at this time.

#### Code Snippet - Detecting Available Scanners

```
int notifications_mask = 0;

// We would like to subscribe to all scanner available/not-available events
notifications_mask |=
    DCSSDKDefs.DCSSDK_EVENT.DCSSDK_EVENT_SCANNER_APPEARANCE.value |
    DCSSDKDefs.DCSSDK_EVENT.DCSSDK_EVENT_SCANNER_DISAPPEARANCE.value;

// We would like to subscribe to all scanner connection events
notifications_mask |=
    DCSSDKDefs.DCSSDK_EVENT.DCSSDK_EVENT_SESSION_ESTABLISHMENT.value |
    DCSSDKDefs.DCSSDK_EVENT.DCSSDK_EVENT_SESSION_TERMINATION.value;

// We would like to subscribe to all barcode events
notifications_mask |= DCSSDKDefs.DCSSDK_EVENT.DCSSDK_EVENT_BARCODE.value;

// enable scanner detection
Application.sdkHandler.dcssdkEnableAvailableScannersDetection(true);

// subscribe to events set in notification mask
Application.sdkHandler.dcssdkSubscribeForEvents(notifications_mask);
```

When the `sdkHandler` calls `dcssdkEnableAvailableScannersDetection(true)`, this enables notifications of type `DCSSDK_EVENT_SCANNER_APPEARANCE`, and `DCSSDK_EVENT_SCANNER_DISAPPEARANCE`. This allows the application to react to a scanner becoming available for connection, discussed in [Connecting to an Available Scanner](#).

## Connecting to an Available Scanner

### Synchronous Scanner Retrieval

There are two ways an application can become aware of an available scanner. The simplest method is to query the SDK (ultimately the underlying Android host) for the scanners that are currently paired and available. This is the synchronous method. This is performed by asking the SDK to pull its current list of available scanners which it retrieves internally upon initialization.

### Code Snippet - Synchronous Scanner Retrieval

```
// ...

    if (Application.sdkHandler != null) {
        mScannerInfoList.clear();
        Application.sdkHandler.dcssdkGetAvailableScannersList(mScannerInfoList);
        Application.sdkHandler.dcssdkGetActiveScannersList(mScannerInfoList);
    }

// ...
```

As shown above, the `sdkHandler` API has two methods:

- `dcssdkGetAvailableScannersList`
- `dcssdkGetActiveScannersList`

These differ in that the `dcssdkGetAvailableScannersList` method retrieves only the scanner devices that are available (i.e., were paired with the Android host). This is different than an established connection. The method `dcssdkGetActiveScannersList` retrieves the scanners (in this version, it is always only one device) that currently has an active connection/session to the SDK library.

### Asynchronous Scanner Notification

When a scanner becomes available, the application is notified through the `dcssdkEventScannerAppeared` event via the `SdkApiDelegate` (in the case of the app, this is the `BaseActivity` class) as shown below.

### Code Snippet - Event Based Available Scanner

```
/* notify connections delegates */
public void dcscdkEventScannerAppeared(DCSSScannerInfo availableScanner) {

    // ...
    // ... Code to update UI delegates and internal structures with
    // ... new scanner data
    // ...

}
```

As a result of the notification from the SDK method `dcscdkEventScannerAppeared`, the application can process the available scanner information how it needs to (i.e., update UI component delegates and internal data structures).

## Performing the Connection

In order to perform the connection, the scanner data must be retrieved either synchronously or asynchronously using the methods described above. Now that the available devices are known, the client application can decide which one it needs to connect to. As with an UI command the actual work should be put into a background task as this allows the Main UI thread to proceed. For the purpose of this example we include this code as well.

### Code Snippet - Scanner Connection

```
public void connectToScanner(View view){
    new MyAsyncTask(scannerId).execute();
}
private class MyAsyncTask extends AsyncTask<Void,Integer,Boolean> {
    private int scannerId;
    public MyAsyncTask(int scannerId){
        this.scannerId=scannerId;
    }
    @Override
    protected void onPreExecute() {
        super.onPreExecute();
        progressDialog = new CustomProgressDialog(AvailableScannerActivity.this,
            "Connect To scanner...");
        progressDialog.show();
    }

    @Override
    protected Boolean doInBackground(Void... voids) {
        DCSSDKDefs.DCSSDK_RESULT result =
            DCSSDKDefs.DCSSDK_RESULT.DCSSDK_RESULT_FAILURE;

        if (Application.sdkHandler != null) {
            result =
                Application.sdkHandler.dcssdkEstablishCommunicationSession(scannerId);
        }

        if(result == DCSSDKDefs.DCSSDK_RESULT.DCSSDK_RESULT_SUCCESS){
            return true;
        }
        else if(result == DCSSDKDefs.DCSSDK_RESULT.DCSSDK_RESULT_FAILURE) {
            return false;
        }
        return false;
    }

    @Override
    protected void onPostExecute(Boolean b) {
        super.onPostExecute(b);
        if (progressDialog != null && progressDialog.isShowing())
            progressDialog.dismiss();
        Intent returnIntent = new Intent();
        if(b){
            setResult(RESULT_OK, returnIntent);
            returnIntent.putExtra(Constants.SCANNER_ID, scannerId);
        }
        else{
            setResult(RESULT_CANCELED, returnIntent);
        }
        AvailableScannerActivity.this.finish();
    }
}
```

The additional code to handle to UI task was include for brevity, however the main SDK API being used is the `dcssdkEstablishCommunicationSession` method which provides the bulk of the connection logic. The result is either success or failure using the result codes defined in the `DCSSDKDefs` class of enums. A result value of `DCSSDKDefs.DCSSDK_RESULT.DCSSDK_RESULT_SUCCESS` indicates that the scanner connection was successful and can be considered accessible.

---

## Receiving Bar Code Data

In order to receive a bar code, first subscribe to the bar code as shown in [Subscribing to Events on page 2-3](#). After completing the event subscription, a class can either directly implement the `IDcsSdkApiDelegate` or inherit it in a way as follows, for example.

### Code Snippet - Processing Bar Code Event

```
public class BaseActivity extends ActionBarActivity implements IDcsSdkApiDelegate {

    // ...

    @Override
    public void dcssdkEventBarcode(byte[] barcodeData, int barcodeType, int
        fromScannerID) {

        Barcode barcode = new Barcode(barcodeData, barcodeType, fromScannerID);
        dataHandler.obtainMessage(Constants.BARCODE_RECIEVED, barcode).sendToTarget();

        // ...
        // Perform other kinds of notifications (i.e. Intents, etc.)
        // ...

    }
}
```

This example creates a new `Barcode` object from the event data itself, which is used to create the bar code message.

The example also uses the Android Message Handler framework to create a message that places the bar code message onto the UI thread using `sendToTarget()`.



**NOTE** The code described above leverages the Android Handler Framework to provide a decoupling from SDK event to the UI thread. For more information on this kind of background to UI thread management, refer to <http://developer.android.com/training/multiple-threads/communicate-ui.html>.

## Retrieving Scanner Attributes

By retrieving scanner attributes, the developer can provide the client application with all the necessary information required from the scanner. The argument format for performing an RSM GET Command uses the following syntax.

### Code Snippet - RSM GET XML Syntax

```
<inArgs>
  <scannerID>ScannerID_Value</scannerID>
  <cmdArgs>
    <arg-xml>
      <attrib_list>attr1,attr2,...,attrN</attrib_list>
    </arg-xml>
  </cmdArgs>
</inArgs>
```

The following example shows the application retrieving standard scanner asset information: *Model Number*, *Serial Number*, *Firmware Version String*, the *Configuration Name* from 123Scan<sup>2</sup>, and the *Date of Manufacturing*. Each of these values is a separate attribute as described previously in the syntax to retrieve.

### Code Snippet - RSM GET Example

```
private void fetchAssertInfo() {

    // get current Scanner ID
    int scannerID = getIntent().getIntExtra(Constants.SCANNER_ID, -1);

    if (scannerID != -1) {

        // Creating beginning of XML argument string with Scanner ID
        // of Active Scanner
        String in_xml = "<inArgs><scannerID>" + scannerID
            + " </scannerID><cmdArgs><arg-xml><attrib_list>";

        // Add attribute values to list
        in_xml+=RMD_ATTR_MODEL_NUMBER;
        in_xml+=",";
        in_xml+=RMD_ATTR_SERIAL_NUMBER;
        in_xml+=",";
        in_xml+=RMD_ATTR_FW_VERSION;
        in_xml+=",";
        in_xml+=RMD_ATTR_CONFIG_NAME;
        in_xml+=",";
        in_xml+=RMD_ATTR_DOM;
        in_xml += "</attrib_list></arg-xml></cmdArgs></inArgs>";

        // Run as Async Task to free up UI
        new MyAsyncTask(scannerID,
            DCSSDKDefs.DCSSDK_COMMAND_OPCODE.DCSSDK_RSM_ATTR_GET).execute(
            new String[]{in_xml});
    } else {
        // Do not have a valid scanner ID, show popup error
        Toast.makeText(this, Constants.INVALID_SCANNER_ID_MSG,
            Toast.LENGTH_SHORT).show();
    }
}
```



In this example, the `OPCODE` defined in `DCSSDKDefs.DCSSDK_COMMAND_OPCODE` is set to `DCSSDK_RSM_ATTR_GET` which instructs the SDK Library to retrieve the values specified, which are:

- `RMD_ATTR_MODEL_NUMBER` - Scanner model number
- `RMD_ATTR_SERIAL_NUMBER` - Scanner serial number
- `RMD_ATTR_FW_VERSION` - Scanner firmware version
- `RMD_ATTR_CONFIG_NAME` - Scanner configuration filename (if one was used)
- `RMD_ATTR_DOM` - Scanner date of manufacture.

Ultimately this calls the interface `executeCommand` which is defined by the `ScannerAppEngine` interface and extended by `BaseActivity`. Ultimately the call to the SDK API `dcssdkExecuteCommandOpCodeInXMLForScanner` is made, which is part of the SDK Handler object owned by the `Application` class as discussed earlier. This is the API call that sends the command to the SDK Library to be queued for communication with the scanner.

### Code Snippet - `dcssdkExecuteCommandOpCodeInXMLForScanner` API

```
public boolean executeCommand(DCSSDKDefs.DCSSDK_COMMAND_OPCODE opCode, String
    inXML, StringBuilder outXML, int scannerID) {

    if (Application.sdkHandler != null)
    {
        // get result from Scanner (not that this is a blocking call, but is
        // being run on an AsyncTask away from the UI thread

        DCSSDKDefs.DCSSDK_RESULT result =
            Application.sdkHandler.dcssdkExecuteCommandOpCodeInXMLForScanner(
                opCode, inXML, outXML, scannerID);

        // check result and return true or false

        if(result == DCSSDKDefs.DCSSDK_RESULT.DCSSDK_RESULT_SUCCESS)
            return true;
        else if(result == DCSSDKDefs.DCSSDK_RESULT.DCSSDK_RESULT_FAILURE)
            return false;
    }
    return false;
}
```

In this example, the call is made directly to the SDK API which sends the opcode out to the scanner. Each attribute is retrieved in turn until they are all retrieved and a valid response is received from the scanner for each one. The API call compiles the results into the `outXML` `StringBuilder` argument for return to the caller. The syntax of the output is as follows:

**Code Snippet - RSM GET XML Syntax**

```

<?xml version="1.0" encoding="UTF-8"?>
<outArgs>
  <scannerID>scanner_ID</scannerID>
  <arg-xml>
    <modelnumber>model_number_value</modelnumber>
    <serialnumber>serial_number_value</serialnumber>
    <response>
      <opcode>DCSSDK_RSM_ATTR_GET</opcode>
      <attrib_list>
        <attribute>
          <id>attribut1</id>
          <datatype>datatype_value</datatype>
          <permission>permission_value</permission>
          <value>attribut1_value</value>
        </attribute>
        <attribute>
          <id>attribute2</id>
          <datatype>datatype_value</datatype>
          <permission>permission_value</permission>
          <value>attribut2_value</value>
        </attribute>
        <attribute>
          <id>attributeN</id>
          <datatype>datatype_value</datatype>
          <permission>permission_value</permission>
          <value>attributN_value</value>
        </attribute>
      </attrib_list>
    </response>
  </arg-xml>
</outArgs>

```

The actual elements returned for each attribute describe the attribute in its entirety:

- id - The Attribute number for which the next elements describe as shown in attributeN
- datatype - One of the following 11 data types.
  - B - Byte - unsigned char
  - C - Char - signed byte
  - F - Bit Flags
  - W - WORD - short unsigned integer (16 bits)
  - I - SWORD - short signed integer (16 bits)
  - D - DWORD - long unsigned integer (32 bits)
  - L - SDWORD - long signed integer (32 bits)
  - A - Array
  - S - String
  - X - Action
- permission - The permission of the data itself; a combination of one or more of the following letters:
  - W - Write - Attribute value is writable
  - R - Read - Attribute value is readable
  - P - Persistent - Attribute value is non-volatile and persists across reboots
- value - The actual value of the attribute. This corresponds to the datatype (e.g., the value for an 'S' datatype is a 16 bit signed number only)

Following is the XML result of the RSM GET command as used in this example (getting all asset information):

### Code Snippet - RSM GET Return Set

```
<?xml version="1.0" encoding="UTF-8"?>
<outArgs>
  <scannerID>6</scannerID>
  <arg-xml>
    <modelnumber>iPL3307-RFD8500</modelnumber>
    <serialnumber>150209008500B </serialnumber>
    <response>
      <opcode>DCSSDK_RSM_ATTR_GET</opcode>
      <attrib_list>
        <attribute>
          <id>533</id>
          <datatype>S</datatype>
          <permission>R</permission>
          <value>iPL3307-RFD8500 </value>
        </attribute>
        <attribute>
          <id>534</id>
          <datatype>S</datatype>
          <permission>R</permission>
          <value>150209008500B </value>
        </attribute>
        <attribute>
          <id>20004</id>
          <datatype>S</datatype>
          <permission>R</permission>
          <value>PAABLS00-004-R00 </value>
        </attribute>
        <attribute>
          <id>616</id>
          <datatype>S</datatype>
          <permission>RWP</permission>
          <value>Modified </value>
        </attribute>
        <attribute>
          <id>535</id>
          <datatype>S</datatype>
          <permission>R</permission>
          <value>15May15</value>
        </attribute>
      </attrib_list>
    </response>
  </arg-xml>
</outArgs>
```

## Sending Remote Commands

The `Attribute SET` mechanism allows a client application to perform many command and control functions on the scanner, from disabling a specific bar code symbology (F - Flag Attribute) to rebooting the device (X - Action Attribute). Following are examples that illustrate how to do this. For a specific product, refer to the device Product Reference Guide to determine what attributes the device supports and the values for that device.

### Beep the Beeper

A simple command example is to sound the beeper on the scanner (if it has one). The following example includes the application UI portion to show how the `Async Task` can process this. For the `executeCommand` function see [Code Snippet - `dcssdkExecuteCommandOpCodeInXMLForScanner API` on page 2-9](#).

#### Code Snippet - Beep the Beeper

```
public void beeperAction(View view) {

    // set beeper to perform a HIGH pitch SHORT duration tone
    int value = RMD_ATTR_VALUE_ACTION_HIGH_SHORT_BEEP_1;

    String inXML = "<inArgs><scannerID>" + getIntent().getIntExtra(
        Constants.SCANNER_ID, 0) + "</scannerID><cmdArgs><arg-int>" +
        + Integer.toString(value) + "</arg-int></cmdArgs></inArgs>";

    // Execute in an AsyncTask to remove UI blocking
    new MyAsyncTask(getIntent().getIntExtra(Constants.SCANNER_ID,
        0),DCSSDKDefs.DCSSDK_COMMAND_OPCODE.DCSSDK_SET_ACTION).execute(
        new String[]{inXML});
}

private class MyAsyncTask extends AsyncTask<String,Integer,Boolean> {
    int scannerId;
    DCSSDKDefs.DCSSDK_COMMAND_OPCODE opcode;

    public MyAsyncTask(int scannerId, DCSSDKDefs.DCSSDK_COMMAND_OPCODE opcode){
        this.scannerId=scannerId;
        this.opcode=opcode;
    }

    @Override
    protected void onPreExecute() {
        super.onPreExecute();
        progressDialog = new CustomProgressDialog(BeeperActionsActivity.this,
            "Executing beeper action..");
        progressDialog.show();
    }

    @Override
    protected Boolean doInBackground(String... strings) {
        return executeCommand(opcode, strings[0], null, scannerId);
    }
}
```

(continued on next page)

```

@Override
protected void onPostExecute(Boolean b) {
    super.onPostExecute(b);
    if (progressDialog != null && progressDialog.isShowing()) {
        progressDialog.dismiss();
    }
    if(!b){
        Toast.makeText(BeeperActionsActivity.this,
            "Cannot perform beeper action", Toast.LENGTH_SHORT).show();
    }
}
}

```

This example is not specifically concerned with the return XML string because it is just requesting an action to be carried out (which is why a null value is passed into `executeCommand()`). The example requests setting the beeper command to `RMD_ATTR_VALUE_ACTION_HIGH_SHORT_BEEP_1` which is a specific pitch and tone (typically this would use the value set by the UI picker, but this example uses a specific value for brevity). The SDK API takes the rest of the underlying XML and communication to cause the beeper to sound.

The result for a command like this is returned via the `onPostExecute` argument Boolean `b` which is `true` on success; `false` otherwise.

## Disabling a Bar Code Symbology Type

The client application can control symbologies on the scanner and determine whether it decodes the supported symbology. This example disables the UPC-A symbology so the scanner can not decode it. This example does not include the UI Async Task code, but focuses on the construction of the XML arguments.

### Code Snippet - Disable UPC-A Example

```

// ...
// UI code removed
// explicitly setting to UPC-A to false for example
// This is Attribute '1' - see XML string below
// ...

String inXML =
    "<inArgs>" +
        "<scannerID>" + getIntent().getIntExtra(Constants.SCANNER_ID, 0) +
        "</scannerID>" +
        "<cmdArgs>" +
            "<arg-xml>" +
                "<attrib_list><attribute>" +
                    "<id>1</id>" +
                    "<datatype>F</datatype>" +
                    "<value>False</value>" +
                "</attribute></attrib_list>" +
            "</arg-xml>" +
        "</cmdArgs>" +
    "</inArgs>";

// ...
// UI code removed
// We are going to use Attribute STORE to make change permanent
// so it persists through scanner reboots
// ...

// Execute Async Task to remove from UI thread
new MyAsyncTask(getIntent().getIntExtra(Constants.SCANNER_ID, 0),
    DCSSDKDefs.DCSSDK_COMMAND_OPCODE.DCSSDK_RSM_ATTR_STORE, view).execute(
    inXML);

```

This example explicitly sets the values for UPC-A and false. Typically these values are retrieved from the UI settings (refer to the application for more details). In the XML string, the example gets the scanner ID from the UI and sets the ID to '1' which is Attribute 1. The datatype is set to 'F' as the UPC-A setting is a Flag datatype and accepts either 'True' or 'False' as an acceptable value. Once the XML string is set, a call is made using the `AsyncTask` defined in the `SymbologiesActivity` class and the opcode `DCSSDKDefs.DCSSDK_COMMAND_OPCODE.DCSSDK_RSM_ATTR_STORE` is passed in as this is an Attribute Store operation. If the setting is temporary and to be restored to its default state after a scanner reboot, use the opcode `DCSSDKDefs.DCSSDK_COMMAND_OPCODE.DCSSDK_RSM_ATTR_SET`.

## Disabling the Scanner

### Code Snippet - Disable the Scanner

Certain use cases require disabling scanning in the scanner, and then re-enabling it when needed. As with most operations, this involves using an opcode and the associated XML argument string.

```
public void disableScanning(View view) {

    String in_xml = "<inArgs><scannerID>" + scannerID + "</scannerID></inArgs>";
    new MyAsyncTask(
        scannerID,DCSSDKDefs.DCSSDK_COMMAND_OPCODE.DCSSDK_DEVICE_SCAN_DISABLE).
        execute(new String[]{in_xml});
}

// ...
// AsyncTask code
// ...

@Override
protected Boolean doInBackground(String... strings) {

    if (Application.sdkHandler != null)
    {

        // calling execute command SDK API
        DCSSDKDefs.DCSSDK_RESULT result =
            Application.sdkHandler.dcssdkExecuteCommandOpCodeInXMLForScanner(
                opCode,inXML,outXML,scannerID);

        // return true if DCSSDKDefs.DCSSDK_RESULT.DCSSDK_RESULT_SUCCESS
        // false otherwise
        if(result== DCSSDKDefs.DCSSDK_RESULT.DCSSDK_RESULT_SUCCESS) {
            return true;
        }
        else if(result==DCSSDKDefs.DCSSDK_RESULT.DCSSDK_RESULT_FAILURE)
            return false;
    }
    return false;
}
```

This example makes a call to `dcssdkExecuteCommandOpCodeInXMLForScanner` using opcode `DCSSDKDefs.DCSSDK_COMMAND_OPCODE.DCSSDK_DEVICE_SCAN_DISABLE`. This disables the scanner so it can not illuminate and scan bar codes. To re-enable the scanner, use opcode `DCSSDKDefs.DCSSDK_COMMAND_OPCODE.DCSSDK_DEVICE_SCAN_ENABLE`.

Most operations performed with the SDK API involve passing opcodes and XML arguments to the `dcssdkExecuteCommandOpCodeInXMLForScanner` method. For a list of all of available opcodes, refer to the included JavaDocs under the `DCSSDKDefs.DCSSDK_COMMAND_OPCODE` enum.





Zebra Technologies Corporation  
Lincolnshire, IL U.S.A.  
<http://www.zebra.com>

Zebra and the stylized Zebra head are trademarks of ZIH Corp., registered in many jurisdictions worldwide. All other trademarks are the property of their respective owners.

© 2016 Symbol Technologies LLC, a subsidiary of Zebra Technologies Corporation. All rights reserved..

