Honeywell

SwiftDecoder™ SDK

For Android™

Software Integration Manual

Disclaimer

Honeywell International Inc. and its affiliates, subsidiaries, and other entities forming part of Honeywell group (“HII”) reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult HII to determine whether any such changes have been made. The information in this publication does not represent a commitment on the part of HII.

Honeywell warrants goods of its manufacture as being free of defective materials and faulty workmanship during the applicable warranty period. Honeywell’s standard product warranty applies unless agreed to otherwise by Honeywell in writing; please refer to your order acknowledgment or consult your local sales office for specific warranty details. If warranted goods are returned to Honeywell during the period of coverage, Honeywell will repair or replace, at its option, without charge those items that Honeywell, in its sole discretion, finds defective. **The foregoing is buyer’s sole remedy and is in lieu of all other warranties, expressed or implied, including those of merchantability and fitness for a particular purpose. In no event shall Honeywell be liable for consequential, special, or indirect damages.** While Honeywell may provide application assistance personally, through our literature and the Honeywell web site, it is buyer’s sole responsibility to determine the suitability of the product in the application. Specifications may change without notice. The information we supply is believed to be accurate and reliable as of this writing. However, Honeywell assumes no responsibility for its use.

Web Address: automation.honeywell.com

For patent information, refer to www.hsmpats.com.

Customer Support and Technical Assistance

For customer support, contact your local Honeywell Sales Representative or fill out the support form at sensing.honeywell.com/contact-support-form.

For our latest contact information, see sensing.honeywell.com/contact.

TABLE OF CONTENTS

CHAPTER

1 INTRODUCTION

Purpose

The Software Integration Guide for SwiftDecoder™ SDK contains information on how to successfully integrate SwiftDecoder technology into your Android application.

Scope

This Software Integration Guide contains information for the following software components of the SwiftDecoder SDK software solution.

- SwiftDecoder Core Barcode Decoding Package (aar though SFTP download or maven package download from GitHub)
- SwiftOCR Package (only maven package download from github) depends on core
- Drivers licesne package (only maven package download from github) depends on core
- API License Key
- Template QR info if using SwiftOCR package in TemplateOCR mode

Intended Audience

The intended audience is a software developer looking to understand the integration steps needed to integrate SwiftDecoder technology into an application.

Terms, Acronyms, & Abbreviations

Below are the terms, acronyms, and abbreviations used within this document. Additional project-specific terms can be found in the project glossary.

Term, Acronym, Abbreviation	Definition
API	Application Programming Interface
SDK	Software Development Kit

CHAPTER

2 DESIGN CONSIDERATIONS AND API UPDATES

Maven Packages (from 6.0)

From 6.0 release onwards the Android packages will be available on private maven repo. See below for the maven references.

Refer sample app for quick start guide.

Build.gradle changes for using maven packages.

```
repositories {
    maven {
        // Reference the repositories defined in the root project
        url = uri("https://maven.pkg.github.com/HON-IA-SD/
swiftdecoder-android")
        credentials {
            username = project.findProperty("gpr.user") ?:
System.getenv("USERNAME_GITHUB")
            password = project.findProperty("gpr.token") ?:
System.getenv("TOKEN_GITHUB")
        }
    }
}
dependencies {
    implementation 'com.honeywell:swiftdecoder:6.0.x'
    implementation 'com.honeywell:swiftocr:6.0.x.x' // if using
swiftocr feature
   implementation 'com.honeywell:swiftdecoderdl:6.0.x' //if using
dl feature
}
```

As it is a private package, it needs the username and token which can be commonly provided in the gradle.properties file as shown below.

The token needs to be generated through the developer option on settings for a given user.

```
gpr.user=HON-IA-SD
gpr.token=<token >
gpr.packages.pkgUsername=HON-IA-SD
gpr.packages.pkgPassword=<token >
```

Decoding Options

With SwiftDecoder, barcode scanning can be achieved several different ways.

- By calling scanBarcode() on an HSMDecoder instance
- By embedding an HSMDecodeComponent within your own activity
- By writing your own SwiftPlugin

scanBarcode

The first, and simplest, is by calling scanBarcode() on an HSMDecoder instance. This will launch the default barcode scanning activity (HSMCameraPreview) and will return the result to any listeners.

HSMDecodeComponent

Another option is to embed an HSMDecodeComponent within your own activity. An HSMDecodeComponent is the real-time camera preview frame layout that can be included in your own activity for greater control over the look and feel of the barcode scanning operation. This is the component that is used behind the scenes in the HSMCameraPreview activity.

SwiftPlugin

The last option is to write your own SwiftPlugin to modify the barcode scanning experience even further. This option allows you to register your plug-in with the system and completely control the look and function of a barcode scanning operation. Any registered plug-ins will be run in both the default HSMCameraPreview activity as well as within any HSMDecodeComponent.

OCR Decoding Option (from 6.0)

This decoding option allows you to scan and decode any generic OCR text found on labels, receipts, forms, etc. This feature is available on SwiftOCR package as a microservice for each of the detection modes.

The 2 detection modes, TEMPLATE_OCR and OPEN_OCR, can be set using setSwiftOCRDetectionMode API and Scan needs to be performed using the core package HSMDecodeComponent.

Redundancy Check Option (from 6.1)

The Redundancy feature, if enabled, provides an option to set how many consecutive redundant barcode results are achieved before the result is transmitted.

Drivers License Decoding Option (from 6.0)

This decoding option allows you to scan and decode any International Drivers license which returns decoded data in EZDL format.

This feature is available on DL package as a microservice.

See SDM Android SDK Components beginning on page 11 for more information.

Image Resolution Configuration API (from 6.0)

From 6.0 release onwards the core package handling camera allows you to configuring the image resolution to be used for decoding. Refer to the API document for further details on the API parameters and return values. Here is the quick reference of the API's.

Resolutions	getCurrentCameraResolution ()
ArrayList< Resolutions>	getSupportedCameraResolutions ()
boolean	setCameraResolutions (Resolutions resolutions)

Augmented Reality Feature changes (from 6.0)

Augmented Reality feature improvements have been added on 6.0.

Apart from improvements, a new plugin called SwiftFindPlugin has been added which allows search and find functionality on barcodes. Existing integrators need to look for compilation issues linked with the below renaming of existing plugins.

- Existing AugmentedRealityPlugin renamed to PreviewSelectPlugin and moved to com.honeywell.plugins.ar.previewselect
- PanaromicDecodePlugin renamed to BatchScanPlugin and moved to com.honeywell.plugins.ar.batchscan
- PanaromicDecodeResultListener renamed to BatchScanResultListener and moved to com.honeywell.plugins.ar.batchscan
- StockPlugin moved to com.honeywell.plugins.ar.stockscan
- New API called getPreviewResult added on PreviewSelectPlugin
- New API called getScanBatchResult added on BatchScanPlugin

Android Minimum and Target SDK Version Updates

For increased security, we recommend using targets with minimum Android SDK version 29 and above as Google has stopped support for earlier Android SDK versions.

SwiftDecoder SDK Version	Min. Android SDK Version	Target Android SDK Version
v 6.0	v 29	v 34

We allow our customers greater flexibility for older devices by setting the minimum requirement lower in the SwiftDecoder SDK, however end application developers are advised to follow best practices as indicated by Google to avoid any security issues.

Camera and Camera 2 API (from 5.3)

With SwiftDecoder versions 5.3 and above, the Android Camera2 API is now the default camera API used in the SwiftDecoder SDK. HSMDecoder allows the user to control the camera properties manually by using Camera2's CaptureRequest class by calling setCaptureRequestBuilderListener(...).

Note: *Android Developer Documentation is available at https://developer.android.com/reference/android/hardware/camera2/CaptureRequest*

Example:

```
hsmDecoder.setCaptureRequestBuilderListener(new
CaptureRequestBuilderListener() {
   @Override
   public void OnCaptureRequestBuilderAvailable(CaptureRequest.Builder
captureRequestBuilder) {
      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
         captureRequestBuilder.set(CaptureRequest.CONTROL_MODE,
            CameraMetadata.CONTROL_MODE_OFF);
         captureRequestBuilder.set(CaptureRequest.CONTROL_AE_MODE,
            CameraMetadata.CONTROL_AE_MODE_OFF);
         captureRequestBuilder.set(CaptureRequest.CONTROL_AF_MODE,
            CameraMetadata.CONTROL_AF_MODE_CONTINUOUS_PICTURE);
         captureRequestBuilder.set(CaptureRequest.SENSOR_SENSITIVITY,
            400);
         captureRequestBuilder.set(CaptureRequest.SENSOR_EXPOSURE_TIME,
            (long) 41666666);
      }
   }
}
```

EZDL and Parser Library Integration (from 5.5)

Starting in the 5.5 release we have integrated EZDL as part of the SwiftDecoder SDK. We also have included various parsers inside the library for Motor Vehicle, Boarding Pass, and Passport/Visa/ID scanning.

- **MRZ** — Machine readable zone at the bottom of travel identification documents such as a Passport, Visa, or ID card
- **EZDL** — PDF417 on the back of driver's licenses in the United States and Canada
- **EZBP** — PDF417 on flight boarding passes
- **EZMV** — PDF417 on motor vehicle documents such as the Title and Registration.

API modification for EZDL

Please modify your application if using a separate EZDL library. The library is now included in SwiftDecoder.

1. Check that your license key contains the EZDL Microservice. If you do not know if your license contains EZDL, please check using the API call ***isLicenseEnabled()*** or contact your technical representative.
2. Modify the EZDL namespace from ***com.honeywell.easydl*** to ***com.honeywell.parser***
3. The interface ***LicenseParser.parseRawData*** no longer needs context parameter to be passed.

CHAPTER

3 HANDLING UNIQUE IDS FOR LICENSING

In Android mobile platforms SDK, there are 2 options for Unique ID:

- Serial number
- ANDROID_ID

Only one option can be used at a time.

Once the device is activated with a specific option, changing that without returning the license will cause double activation for same.

App Type	Unique ID	Constraints	SDK API	Reference Link
Built-in Apps	**Serial Number**: The device's serial number is a unique identifier assigned by the manufacturer.	Integrating application needs READ_PRIVILEGED_PHONE_STATE permission	useSerialNumber is the API available on ActivationManager. Refer API Documentation	https://developer.android.com/reference/android/os/Build#getSerial()
PlayStore Apps	**ANDROID_ID**: The Settings.Secure.ANDROID_ID is a unique identifier for an Android device.	On Android 8.0 (API level 26) and higher versions of the platform • ANDROID_ID is unique to each combination of app-signing key, user, and device • Values of ANDROID_ID are scoped by signing key and user. • The value may change if a factory reset is performed on the device or if an APK signing key changes	If not using serial number this would be the default option used for license activation.	https://developer.android.com/reference/android/provider/Settings.Secure#ANDROID_ID

Note: *For apps that were installed prior to updating the device to a version of Android 8.0 (API level 26) or higher, the value of ANDROID_ID changes if the app is uninstalled and then reinstalled after the OTA. To preserve values across uninstalls after an OTA to Android 8.0 or higher, best approach is to return the license before upgrade and the reactivate after the upgrade operation.*

The table below also gives a brief idea of how combination of app-signing key, user, and device would be handled when ANDROID_ID is used.

Factors for Android_ID	Signing key	User	Device	Conclusion
App A	same	same	same	App A and App B will have Similar Unique ID
App B	same	same	same	
App A	diff	same	same	App A and App B will have different Unique ID
App B	diff	same	same	
App A	same	diff	same	App A and App B will have different Unique ID
App B	same	diff	same	
App A	same	As device is diff, user will be diff	diff	App A and App B will have different Unique ID
App B	same	As device is diff, user will be diff	diff	

CHAPTER

4 SDM ANDROID SDK COMPONENTS

Overview

This section provides details about the main SDK components.

Adding SwiftDecoder Dependency

There are 2 ways to include the SwiftDecoder Dependency for integrating application.

- **SwiftDecoder(.aar)** - The legacy approach through direct dependency on aar (if integrating application is only using SwiftDecoder Core Library).
- **Maven Package** - The new approach if the integrating application is also using SwiftOCR or DL. In this case, all the packages need to be added as a maven package.

SwiftDecoder(.aar) build.gradle

Only for SwiftDecoder core dependency.

```
repositories {
   flatDir {
      dirs 'libs'
   }
}
dependencies {
   implementation(name:'SwiftDecoderMobile', ext:'aar')
}
```

Maven Package

For SwiftDecoder core and other dependent packages.

```
repositories {
   maven {
      // Reference the repositories defined in the root project
      url = uri("https://maven.pkg.github.com/HON-IA-SD/swiftdecoder-
```

```
android")
      credentials {
         username = project.findProperty("gpr.user") ?:
System.getenv("USERNAME_GITHUB")
         password = project.findProperty("gpr.token") ?:
System.getenv("TOKEN_GITHUB")
      }
   }
}
dependencies {
   implementation 'com.honeywell:swiftdecoder:6.0.x'
   implementation 'com.honeywell:swiftocr:6.0.x.x' // if using swiftocr
feature
   implementation 'com.honeywell:swiftdecoderdl:6.0.x' //if using dl
feature
}
```

API License Key & Licensing Models

The API license key will be delivered to the licensee via Honeywell's Secure FTP website. This needs to be passed to the ***ActivationManager.activate*** method before the API will be fully functional. This license key should be kept safe and never given to anyone that does not require access to it.

This should be the first decoding related operation your application performs.

Note: *Any settings made to HSMDecoder before activation has occurred will be overridden upon activation.*

License Key Activation

Before scanning can occur, a license key must be activated. For customers using SDK versions 5.3 and above, please activate your license using the **entitlement ID**.

To activate with your entitlement ID, call the following method for a one-time activation via the web:

```
ActivationManager.activateEntitlement
```

Caution: Previous versions of this library used the activation ID which is now deprecated. We do not recommend using the activation ID based activation:

```
ActivationManager.activate
```

One-Time Remote Activation

This license model ActivationManager.Entitlement() must be called each time an application is first launched, however, the API will only need to contact a remote licensing server once. Once a successful activation has occurred with the remote server, the API will no longer require internet access. However, the

ActivationManager class also contains a method named deactivate() which relinquishes the license back to the license server (requires internet connectivity). If this is called, the device will no longer decode barcodes until ActivationManager.Entitlement() is called and the device can re-acquire a license from the remote license server. These two methods facilitate a floating licensing model, should it be desired. This will only be supported if enabled by your license type.

Please ensure your application will have access to:

https://honeywellsps.flexnetoperations.com/ on **TCP443**

One-Time Local Server Activation

While one-time remote activation is the preferred licensing model, it may not be suitable for all use cases. If your use case will never have internet connectivity (not even once) then we provide a local license server instance that you can run on a dedicated Windows PC within your facility.

In this case, you will be given a LocalLicenseServer_Deliverable.zip file containing a license server that your devices will need connectivity to in order to activate the SwiftDecoder software. Like the one-time remote activation model, a device will only need connectivity to this server once. Within this zip file are instructions on how to install, run, and provision this license server. Each time your application is started it will need to call the ActivationManager.activate(Context context, String licenseKey, String localLicenseServerURL, byte[] identitiyClient) method passing you license key, server URL including port and the contents of IdentityClient.bin (included in zip file).

HSMDecoder

This class is responsible for all decode related functions and configurations. HSMDecoder is a singleton, meaning there only ever exists one instance which can be accessed anywhere within your application. The instance can be obtained by calling the static method HSMDecoder.getInstance() and must be disposed of by calling the static method HSMDecoder.disposeInstance() before the application is closed. This class exists within HSMDecoderAPI.jar in the com.honeywell.barcode namespace.

Note: *HSMDecoder Instance needs to be created only after license activation is success as shown below in the sample code. This sequence needs to be followed strictly. In case integrating application needs to destroy HSMDecoder instance then also same sequence to be followed as part of reinitialization, first license activation and then HSMDecoder instance.*

Example

```
//any class that wishes to receive decode results
//must implement the DecodeResultListener interface
```

```
public class Main extends AppCompactActivity implements
DecodeResultListener
{
   private HSMDecoder hsmDecoder;

   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);

      //activate the API with your license key
      ActivationManager.activateEntitlement(this, "entitlement-id");

      //get the singleton instance of the decoder
      hsmDecoder = HSMDecoder.getInstance(this);

      //register a listener for a default decode result
      //Note: if there is at least one listener for the HSMDecoder class,
the default barcode scanning plug-in
      //will be displayed in both HSMCameraPreview and/or within any
HSMDecodeComponent instance
      hsmDecoder.addResultListener(this);

      //set all decoder related settings
      hsmDecoder.enableSymbology(Symbology.UPCA);
      hsmDecoder.enableSymbology(Symbology.CODE128);
      hsmDecoder.enableSymbology(Symbology.CODE39);
      hsmDecoder.enableSymbology(Symbology.QR);

      hsmDecoder.enableFlashOnDecode(false);
      hsmDecoder.enableSound(false);
      hsmDecoder.enableAimer(true);
      hsmDecoder.setAimerColor(Color.RED);
      hsmDecoder.setOverlayText("Place bar code completely inside
viewfinder!");
      hsmDecoder.setOverlayTextColor(Color.WHITE);
   }

   @Override
   public void onDestroy()
   {
      super.onDestroy();

      //dispose of the decoder instance, this stops the underlying camera
service and releases all associated resources
      HSMDecoder.disposeInstance();
   }

   private void Decode()
   {
      //initiate the scan barcode activity, this shows a real-time camera
preview screen
      hsmDecoder.scanBarcode();
   }
```

```
//callback method that returns the decode results
   @Override
   public void onHSMDecodeResult(HSMDecodeResults[] barcodeData)
   {
      //process decode results
      tvResult.setText("Result: " + barcodeData[0].getBarcodeData());
      tvSymb.setText("Symbology: " + barcodeData[0].getSymbology());
      tvLength.setText("Length: " +
barcodeData[0].getBarcodeDataLength());
      tvDecTime.setText("Decode Time: " +
barcodeData[0].getDecodeTime() + "ms");
   }
}
```

HSMDecodeComponent

An HSMDecodeComponent is a real-time camera preview frame layout that can be included in your own activity for greater control over the look and feel of the barcode scanning operation. This allows you to resize the camera preview as you see fit. This can be used as a replacement for the HSMCameraPreview activity that is launched by the HSMDecoder scanBarcode() method.

An example of where you may wish to use this is within a tabbed activity. Using an HSMDecodeComponent embedded within your own activity allows you to customize the look and feel of the barcode scanning operation.

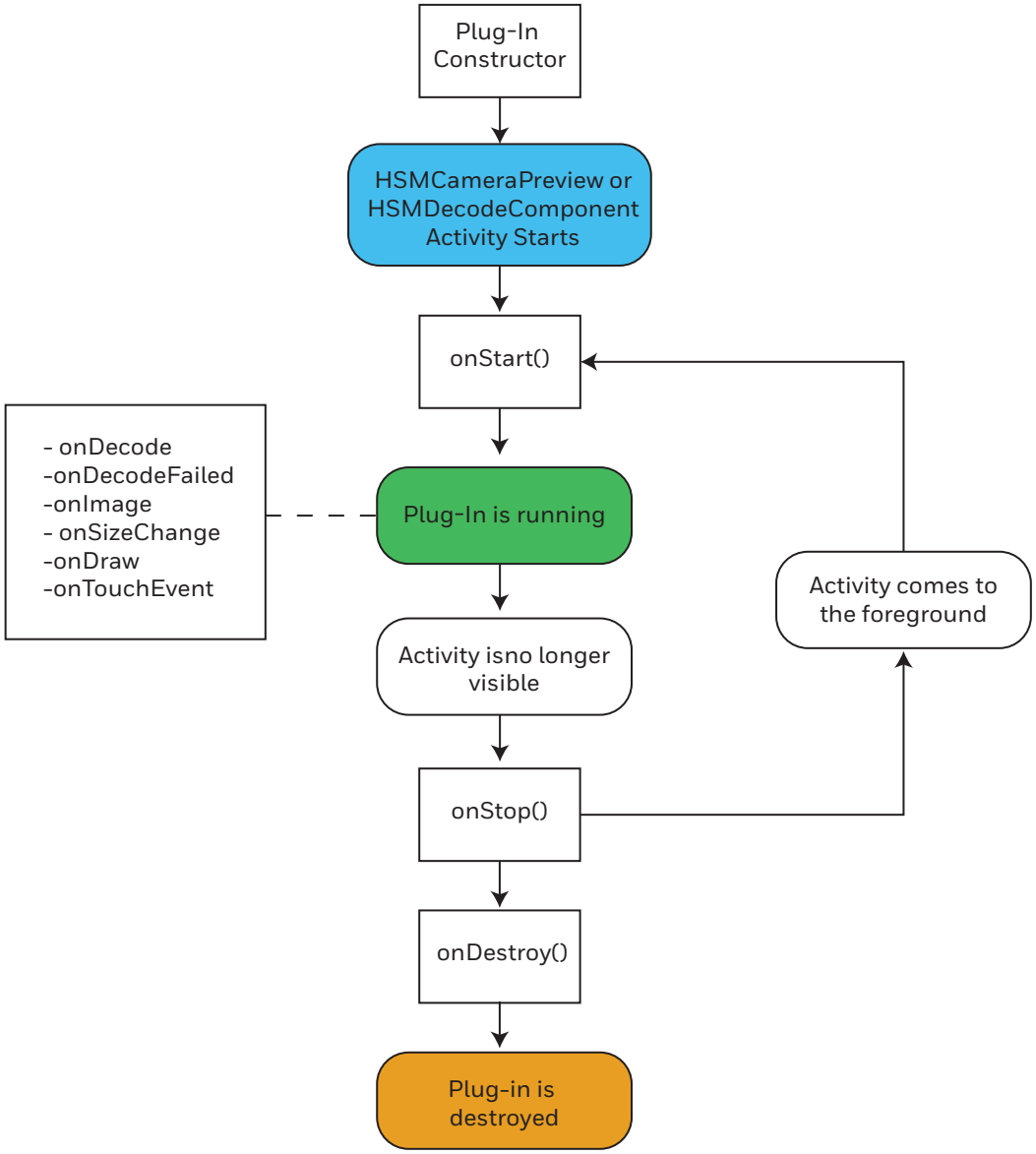
Example

```
<!--custom barcode scanning activity-->
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
   android:layout_width="match_parent"
   android:layout_height="fill_parent">

         <!—define the decode component-->
   <com.honeywell.barcode.HSMDecodeComponent
      android:id="@+id/hsm_decodeComponent"
      android:layout_width="207dp"
      android:layout_height="match_parent" />

</LinearLayout>
```

Plug-In Life Cycle



Plug-In Callbacks

```
//Called when the HSMCameraPreview or HSMDecodeComponent activity comes
to the foreground.
//This should be used to initialize the plug-in on each scan attempt.
protected void onStart(){};

//Called when the HSMCameraPreview or HSMDecodeComponent is no longer
visible
protected void onStop(){};

//Called when the plug-in's dispose() method is called.
//This is used to clean up plug-in resources.
protected void onDestroy(){};

//Called when a barcode(s) has been decoded in the image
protected void onDecode(HSMDecodeResult[] results) {};

//Called each time a frame cannot be successfully decoded
protected void onDecodeFailed(){};

//Called each time an image is sent to the decoder
protected void onImage(byte[] image, int width, int height){};

//Called each time the plug-in's UI size changes. This is called each
time the plug-in is loaded as well.
//This may be used to adjust graphics on the UI
protected void onSizeChanged(int width, int height, int oldWidth, int
oldHeight) {};

//Called each time the plug-in's UI is drawn. The uiLayer is used to
render custom graphics
protected void onDraw(Canvas uiLayer, int width, int height) {};

//Called each time the plug-ins UI is physically touched. Facilitates UI
interaction (and gesturing)
protected void onTouchEvent(MotionEvent event) {};
```

Example

```
//activity that creates a custom plug-in and handles the result
public class Main extends AppCompactActivity implements
MyCustomPluginResultListener
{
   private HSMDecoder hsmDecoder;
   private MyCustomPlugin customPlugin;

   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);

      //activate the API with your license key
      ActivationManager.activateEntitlement(this, "insert-your-
entitlement-api-key-here");

      //get the singleton instance of the decoder
      hsmDecoder = HSMDecoder.getInstance(this);

      //set all decoder related settings
      hsmDecoder.enableSymbology(Symbology.UPCA);
      hsmDecoder.enableSymbology(Symbology.CODE128);
      hsmDecoder.enableSymbology(Symbology.CODE39);
      hsmDecoder.enableSymbology(Symbology.QR);

      //create plug-in instance and add a result listener
      customPlugin = new MyCustomPlugin();
      customPlugin.addResultListener(this);

      //register the plug-in with the system
      hsmDecoder.registerPlugin(customPlugin);
   }

   @Override
   public void onDestroy()
   {
      super.onDestroy();

      //dispose of the decoder instance, this stops the underlying camera
service and releases all associated resources
      HSMDecoder.disposeInstance();

      //dispose of plug-in
   customPlugin.dispose();
   }

   private void Decode()
   {
      //initiate the scan bar code activity, this shows a real-time
camera preview screen and/or runs any custom plug-ins
      hsmDecoder.scanBarcode();
   }
```

```
   //callback method that returns the plug-in result
   @Override
   public void onMyCustomPluginResult (Boolean result)
   {
      //process the plug-in result
   }
}

//interface for the custom plug-in that any "observer" must implement to
receive plug-in results/notifications
//must extend the PluginResultListener interface
public interface MyCustomPluginResultListener extends
PluginResultListener
{
   public void onMyCustomPluginResult(Boolean result);
}

//custom plug-in class, simply returns true to all observers on a
successful decode
public class MyCustomPlugin extends SwiftPlugin
{
   public MyCustomPlugin(){}

   @Override
   protected void onStart()
   {
      super.onStart();
      //do something
   }

   @Override
   protected void onStop()
   {
      super.onStop();
      //do something
   }

   @Override
   protected void onDestroy()
   {
      super.onDestroy();
      //do something
   }

   @Override
   protected void onDecode(HSMDecodeResult[] results)
   {
      super.onDecode(results);

      //tells all plug-in monitor listeners we have a result
      //this is used to signal HSMCameraPreview (if you are using it)
that a result has been found
      //and control should be returned to the caller. This call is not
necessary if you are using an HSMDecodeComponent.
```

```
        this.signalResultFound();

      //notifies all plug-in listeners we have a result
   List<PluginResultListener> listeners = this.getResultListeners();
      for(PluginResultListener listener : listeners)

((MyCustomPluginResultListener)listener).onMyCustomPluginResult(true);
   }

   @Override
   protected void onDecodeFailed()
   {
      super.onDecodeFailed();
      //do something
   }

   @Override
   protected void onImage(byte[] image, int width, int height)
   {
      super.onImage(image, width, height);
      //do something
   }

   @Override
   protected void onSizeChanged(int width, int height, int oldWidth, int
oldHeight)
   {
      super.onUISizeChanged(width, height, oldWidth, oldHeight);
      //do something
   }

   @Override
   protected void onDraw(Canvas uiLayer)
   {
      super.onUIDraw(uiLayer);
      //do something
   }

   @Override
   protected void onTouchEvent(MotionEvent event)
   {
      super.onTouchEvent(event);
      //do something
   }
```

Swift Plugins

A SwiftPlugin is a special java “plugin” class that allows you to completely control the look and function of a barcode scanning operation. All plug-ns have their own UI that is rendered over the real-time camera preview. A SwiftPlugin must extend the SwiftPlugin base class, which contains many callback methods that are fired throughout the plug-in life cycle.

A SwiftPlugin utilizes the observer pattern to notify any “observers” of a decode result (or any notification). Any observer of a SwiftPlugin must register itself as a result listener with the plug-in. This is done by creating an interface that extends the PluginResultListener interface. A SwiftPlugin will be notified when a barcode is decoded, when the screen is drawn, when the screen is touched and many other times throughout the plug-in lifecycle. This allows you to completely control the barcode scanning experience.

You can register your plug-in with the system via the HSMDecoder registerPlugin() method where it will be run in both the default HSMCameraPreview activity, as well as within any HSMDecodeComponent.

Ready to Use Feature Plugins

The following are the set of ready to use feature plugins providing a specific set of functionality.

PreviewSelect Plugin

Used for rendering Augmented Reality Overlay over the detected barcodes within the preview. This plugin mainly allows a "Preview & Select" functionality along with capability to fetch the Preview Overlay count or Result data. The overlays on the preview allow the touch functionality to select one barcode. The application can register the DecodeResultListener call back using HSMDecoder::addResultListener API to have further action on the selected barcode.

BatchScan Plugin

Used for scanning a batch of barcodes, either predefined batch or batch count, until user intervention.

- Option one is with predefined batch limit which ensures that n number of bar codes are decoded before the results are returned to the business logic.
- Option two is to get the batch count would be the n number of barcodes until user intervention using API getScannedBatchResult or getScannedBatchCount.

SwiftFind Plugin

Used for rendering searching and finding barcodes within the preview through overlay indicators.

Windowing/Targeting Plugin

Used to dynamically set the decoding mode using a Window or Target. This plugin mainly allows configuring the "Windowing" /"Targeting" functionality. Setting of the windowing mode is possible through API setWindowMode on this plugin.

When set to Windowing Mode only barcodes within the given window will be decoded. The API enableTouchResizing on this plugin allows resizing of the window. Also it can be set using setWindow API.

When set to Targeting Mode the barcodes on which the target is aimed can be decoded. The size of the target can be set using API setTargetSize on this plugin.

The application can register DecodeResultListener call back using HSMDecoder::addResultListener API to have further action on the selected barcode. Use HSMDecoder::registerPlugin and HSMDecoder::unRegisterPlugin API to register/unregister any plugin on SwiftDecoder.

DLAgeVerificationPlugin

Used to verify a person's age by scanning the barcode on the back of a driver's license issued in the United States or Canada. This plugin allows SwiftDecoder to compare the age from the barcode record against a configured value and indicate if the license holder is above that age. SwiftDecoder displays a check-mark if the age of the ID holder is greater than or equal to the configured age or a cross mark if the age is less than the configured age. The application can register the DLResultListener to display the data from the barcode by tapping the check-mark or cross mark in the app.

PlugIn Details

This is the plugin used to set the age for verification.

```
DLAgeVerificationPlugin
```

The plugin is part of this package:

```
com.honeywell.plugins.ar.DLAgeVerfication
```

Configure the Age Requirement

The age limit can be passed with constructor of the plugin or using the following API:

```
public void DLAgeVerfication::setAgeForVerification(int
ageForVerification)
```

Register the Listener

This is the API used to register the DLResultListener to allow the information in the barcode to be displayed in SwiftDecoder when the user taps the check-mark or cross mark.

```
public void setDLResultListener(DLResultListener dlResultListener)
```

Stock Plugin

This plugin is an extension to Augmented Reality (AR) plugin. It is used to display the stock data returned by the customer as an overlay element.

Initialize Stock Plugin

```
//Create Instance of HSM Decoder
HSMDecoder hsmDecoder
hsmDecoder = HSMDecoder.getInstance(this);

//Create Instance of Stock Plugin
StockPlugin stockPlugin
stockPlugin = new StockPlugin(getApplicationContext());
```

Configuring the image to be used for display on Stock Plugin

```
String imagePath ="/drawable/"+R.mipmap.arrow;
Uri uri = Uri.parse("android.resource://
"+this.getPackageName()+imagePath);
Bitmap bm =
MediaStore.Images.Media.getBitmap(this.getContentResolver(),uri);
ByteArrayOutputStream baos = new ByteArrayOutputStream();
bm.compress(Bitmap.CompressFormat.PNG,100,baos);
byte[] byteArray = baos.toByteArray();
stockPlugin.setImage(Base64.encodeToString(byteArray,Base64.DEFAULT));
```

Configuring Touch Listener

Argument passed to the below API implements public void onResultSelected(ArrayList pluginResult)

For ARPluginTouchListener. Refer API documentation for more details.

```
stockPlugin.setOnARPluginTouchListener(this);
```

Register the plugin

This is the API used to register the plugin.

```
hsmDecoder.registerPlugin(stockPlugin);
```

OnStockResult

This is the API used to send the stock data associated with barcodes to display on the augmented reality user interface.

```
stockPlugin.onStockResult(data);
```

Result Format

```
[
  {
    "barcodeData1": [
      {
        "label1": "label1Value"
      },
      {
        "label2": ""
      },
      {
        "color": "#FF0000"
      }
    ]
  },
  {
    "barcodeData2": [
      {
        "label1": "label1Value"
      },
      {
        "label2": ""
      },
      {
        "color": "#FF0000"
      }
    ]
  }
]
```

Example

```
[
  {
    "12345": [
      {
        "Stock": "42"
      },
      {
        "date": 1651553823831
      },
      {
        "color": "#FF0000"
      }
    ]
  },
  {
    "1234567890": [
      {
        "Stock": "42"
      },
      {
        "date": 1651553823832
      },
      {
        "color": "#FF0000"
      }
    ]
  }
]
```

Freeze Frame Plug-In

Freeze frame is a feature that is enabled in conjunction with preview and select mode. The feature allows the user to capture and image and provide AR overlay within the rendered screen.

Note: *Freeze Frame mode only works when in preview and select mode*

The following sequence is to be followed for using freeze frame with Preview and Select Mode

1. SDK provides an interface named freeze "**setFreezeMode(boolean mode)" on HSMDecodeComponent** to enable/disable the freeze mode.
2. This mode needs be enabled only if registered plugin is AugmentedRealityPlugin.
3. Once the integrating application enables freeze mode the decoding of frames is disabled at the back end. Hence the preview would still be running in freeze mode without decoding any of the frames.
4. On first double tap in the preview screen the captured image frame would be displayed in preview.

5. If the captured image has barcodes, the AR overlay would be rendered on the screen with bounding boxes around the barcodes.
6. User can select the required barcodes with single touch for getting the decoding result. The decoding result is returned as part of **onHSMDecodeResult** implemented by integrating application.
7. With double tap on frozen image the preview screen would continue to capture the video frames.
8. **FreezeFrameListener** call backs to be implemented by integrating application to get the notification for **onFreezeFrame**(first double tap) and **onUnFreezeFrame**(second double tap). Use API **registerFreezeFrameListener(FreezeFrameListener listener)** on HSMDecoder for registering the listener.
9. To know if currently freeze mode is enabled or not use API **isFreezeMode** exposed on HSMDecoder.
10. While changing the freeze mode dispose the old AR plugin first and then register a new AR plugin.
11. In case integrating application needs this feature in full screen mode then can use API **scanBarcodeInFreezeMode** exposed on HSMDecoder.

Note: *For more hands on to understand this feature check the demo using Honeywell Barcode Scanner app available in playstore.*

Parsers

Parsers are separate features that can be enabled within your application to return the parsed data for the raw decoded data. These features are licensed separately from the original standard features. This can be added on at any time. To decode motor vehicles, boarding passes, and driver's license, please ensure that PDF417 barcode symbology is enabled.

Motor Vehicles (EZMV)

Easy Motor Vehicles feature allows to parse the raw decoded data from vehicle documents with PDF417 Barcode. The supported vehicle documents include Title(TD) and Registration(RG).

Refer to the MotorVehicleParser class documentation in the SDK package for more details on API arguments and return values. The MotorVehicleData class documentation describes the list of parsed data fields for each for document type.

Sample Code:

```
import com.honeywell.parser.MotorVehicleParser;
import com.honeywell.parser.MotorVehicleData;
...
if(MotorVehicleParser.isLicenseEnabled()) { //isLicesneEnabled can also
be used once at start/initialization to check instead everytime

   MotorVehicleData mvData = MotorVehicleParser.parseRawData(
results[0].getBarcodeDataBytes());
   if (mvData != null) {

      //Add application logic for using the parsed data object
   }
}
```

Boarding Passes (EZBP)

Easy Boarding Pass feature allows to parse the raw decoded data from boarding passes with PDF417 symbology.

Refer to the BoardingPassParser class documentation in the SDK package for more details on API arguments and return values. The BoardingPassData class documentation describes the list of parsed data fields.

Sample Code:

```
import com.honeywell.parser.BoardingPassData;
import com.honeywell.parser.BoardingPassParser;
...
if(BoardingPassParser.isLicenseEnabled()) { //isLicesneEnabled can also
be used once at start/initialization to check instead everytime
   BoardingPassData boardingPassData =
BoardingPassParser.parseRawData(results[0].getBarcodeDataBytes());
   if (boardingPassData != null) {
      //Add application logic for using the returned parsed data object
   }
}
```

Driver's License (EZDL)

Easy Driver's License feature allows to parse the raw decoded data from Driver's Licenses with PDF417 symbology.

Refer to the LicenseParser class documentation in the SDK package for more details on API arguments and return values. The LicenseData class documentations describes the list of parsed data fields.

Sample Code:

```
import com.honeywell.parser.LicenseData;
import com.honeywell.parser.LicenseParser;
...
if(LicenseParser.isLicenseEnabled()) { //isLicesneEnabled can also be
used once at start/initialization to check instead everytime
   LicenseData licenseData =
LicenseParser.parseRawData(results[0].getBarcodeDataBytes());
   if (licenseData != null) {
      //Add the application logic for using the returned parsed data
object
   }
}
```

Machine Readable Zone (MRZ)

MRZ parser feature allows to parse the raw decoded data for OCR symbology with template passport enabled.

Refer to the MRZParser class documentation in the SDK package for more details on API arguments and return values. The MRZData class documentations describes the list of parsed data fields.

Sample Code:

```
import com.honeywell.parser.MRZData;
import com.honeywell.parser.MRZParser;
...
if(MRZParser.isLicenseEnabled()){ //isLicesneEnabled can also be used
once at start/initialization to check instead everytime

   MRZData mrzData =
MRZParser.parseRawData(results[0].getBarcodeData());
   if(mrzData != null){

      //Add the application logic for using the returned parsed data
object

   }
}
```

OCR-A and OCR-B Font Detection

Optical Character Recognition gives the ability to read OCRA and OCRB fonts. OCR can parse Price Field, ISBN, Passport, and MICR characters. Please refer to the OCR Programming User's Guide (OCR_Honeywell.pdf) document for more information. Custom template information can also be found in the OCR Programming User's Guide.

Sample Code:

```
hsmDecoder.enableSymbology(Symbology.OCR);
hsmDecoder.setOCRActiveTemplate(OCRActiveTemplate.PASSPORT);
```

SwiftOCR

Swift OCR is a new separate package from 6.0 release dependent on the core package mainly allowing OCR functionality.

Below is sample reference for enabling this mode.

Detection mode needs to be set prior to enable the API.

```
SwiftOCRDecoder genericOCRDecoder;
genericOCRDecoder.setOCRDetectionMode(SwiftOCRDecoder.SwiftOCRDetectio
nMode.OPEN_OCR);
SwiftOCRStatus enableStatus =
genericOCRDecoder.enableSwiftOCRFeature(true);
```

Release Mode Settings for Code Obfuscation

The following rules need to be added to ProGuard for the integrating application to avoid any runtime exceptions on SwiftOCR Library in release mode when code obfuscation is enabled:

```
keep class com.honeywell.swiftocr.SwiftOCRTemplateModel{ *;}
keep class com.honeywell.swiftocr.SwiftOCRTemplateModel$Tags{ *;}
```

Template OCR

Template OCR is a feature that allows a users to parse important OCR data on user supplied labels. This is done by passing the template information associated with the label to the SDK. This can be useful in a variety of manners when trying to cross reference information between a barcode and what is printed on the labels/tags. Once user provides a label or tag, Honeywell can generate custom templates provided they meet the following guidelines,

1. There must be a minimum of one barcode on the label
2. Only labels with English characters
3. Apart from English characters, ASCII characters are supported as well

An example of where you might want to use it would be when checking the pricing printed on the shelves with the user data base. By using Template OCR, you can extract the barcode information as well as other information printed on the users tag for price checking. You can use this additional information for a multitude of functions, like price checking.

Interfaces

Before starting to use these interfaces, customers should have submitted their example labels to Honeywell SA along with details of the important OCR detection for Template generation. Please refer to the template generation workflow in below section for more details. Customers developing the APP need to store these QR

code template files shared by Honeywell within the application package. Templates can be tested using the Honeywell Scanning Demo Application to verify the outputs.

The Interfaces can be grouped into the following categories:

- Configuring the templates
- Activating the template
- Getting the OCR result

Template Configurations

Templates can be configured one time during the initialization. A maximum of 40 templates can be configured by a customer application.

Interface	Input	Output
`addOCRTemplate`	bye[] (Android / NSData* (IOS) QRCode Template file data as an array	int - <templateID> : Template ID starting from 1 if template added successfully. -2 : If template was laready added and is a duplicate -3 : If template count goes more than 40 -1 : Any generic error in template addition
`removeOCRTemplateID`	int The ID of Template that was already added	int Input Template ID on SUCCESS of removal -1 ERROR while removing the template ID

Activating the template

Of the 40 templates, only one can be active at a time. Before setting the template ID as active there are 2 API's that need to be used. Follow the sequence below.

	Interface	Input	Output
1	`enableSymbology`	int Symbology.OCR	boolean True if enable SUCCESS False if enable ERROR
2	`setOCRActiveTemplate`	int ADVANCED_TEMPLATE as defined in OCRActiveTemplate	void

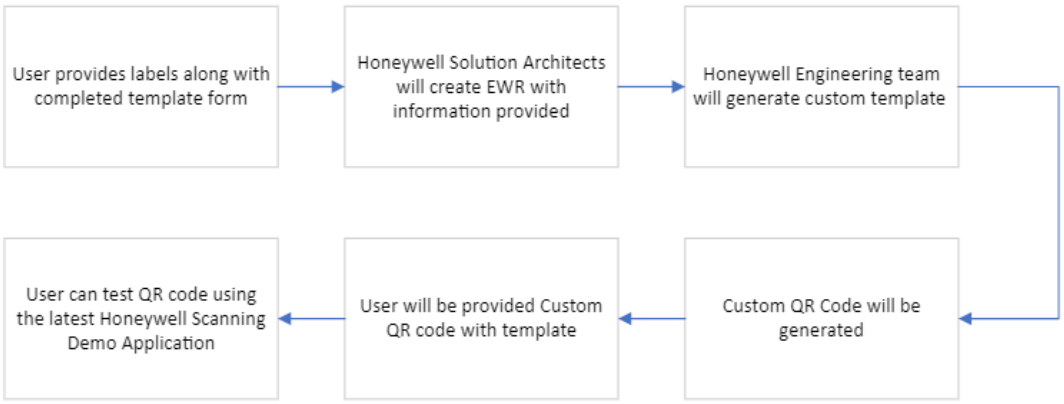
3	`setOCRActiveTemplateID`
	
	`Can be used at runtime to set the active template ID with above 2 APIs already set.`
		int
		
		The intially configured Template ID that needs to be active
			int
			Input template ID that was passed for setting active if SUCCESS
			-1 if ADVANCED_TEMPLATE is not set as an active template
			-2 If input Template ID is not a Valid ID

Getting the OCR Result

Since the OCR detection is related to a barcode within a given label, the results for OCR detection are returned on SwiftOCRResultListener on call back onSwiftOCRTemplateResult where the data is embedded onto SwiftOCRTemplateResult class object. Refer to the API documentation for more details.

Interface	Input	Output
`SwiftOCRTemplateResult`	None	getSwiftOCRDecodeResults Returns an rray of SwiftOCRResults

Template Generation Process



OPEN OCR

Open OCR detection mode allows to detect any general OCR text without any barcode reference as compared to template OCR.

There are 2 options for this mod:

- TARGETED_SINGLE_ROI
- FULL PREVIEW (needs to be set based on the requirement)

TARGETED_SINGLE_ROI

Provides a targeted window within he preview to scanning any targeted text.

```
   SwiftOCRDecoder genericOCRDecoder;
genericOCRDecoder.setSwiftOCRScanArea(SwiftOCRDecoder.SwiftOCRScanArea
.TARGETED_SINGLE_ROI);
```

TARGETED_SINGLE_ROI does allow some more configuration of the targeted window size, orientation etc refer the API document for more details.

FULL PREVIEW

Allows you to scan the text within the given preview window.

```
   SwiftOCRDecoder genericOCRDecoder;

genericOCRDecoder.setSwiftOCRScanArea(SwiftOCRDecoder.SwiftOCRScanArea
.FULL_PREVIEW);
```

OCR result listener to be added using API

```
SwiftOCRDecoder.getInstance(context).addResultListener(this);
```

Getting the OCR Result

The results for OCR detection are returned on SwiftOCRResultListener call back on`SwiftOCRResult`. Refer to the API documentation for more details.

International DL Scan

The International DL scan feature allows you to scan the front of a driver's license which is mainly OCR for most of the countries except for the US which use both barcode and OCR. The feature is available on package 'com.honeywell:swiftdecoderdl'

Refer the SampleDLProject provided along with package for quick start on the API's to be used for integrating this feature.

For enabling DL scan:

DLDecoderStatus status = DLDecoder.getInstance(getApplicationContext()).setDLScanFeature(isDL,getApplicationContext(), dlScanType);

For getting the result implement interface DLResultListener and its callback void onDLDecodeResult(LicenseData licensedata);

Redundancy Check

The Redundancy feature, if enabled, provides an option to set how many consecutive redundant barcode results are achieved before the result is transmitted.

The feature is currently implemented for the following symbologies:

- Code128
- Codabar
- Code39
- EAN13
- EAN8
- UPCA
- Int25
- PDF417
- QR

The feature must be enabled separately for each symbology along with the number of iterations.

For enabling Redundancy:

```
public boolean SetSymbologyRedundancy(String symbology,boolean
isRedundancyEnabled, int vote)
```

For getting the result:

```
public boolean GetSymbologyRedundancy(String symbology, int[] vote)
```

Honeywell
855 S. Mint St.
Charlotte, NC 28202

automation.honeywell.com