CyberSource Mobile Point of Sale

Android Integration Guide

April 2019

CyberSource®
the power of payment

CyberSource Corporation HQ | P.O. Box 8999 | San Francisco, CA 94128-8999 | Phone: 800-530-9095

CyberSource Contact Information

For general information about our company, products, and services, go to http://www.cybersource.com.

For sales questions about any CyberSource Service, email sales@cybersource.com or call 650-432-7350 or 888-330-2300 (toll free in the United States).

For support information about any CyberSource Service, visit the Support Center: http://www.cybersource.com/support

Copyright

Restricted Rights Legends

Trademarks

Recent Revisions to This Document

Release	Changes
April 2019	Added the following sections: ■ Partial Authorization, page 31 ■ Partial Refund, page 42 ■ Registering a Device Using the Sample Application, page 47 ■ Setting Up Multiple User Accounts Using a CSV File, page 47
January 2019	This is the first release of this document.

About This Guide

Welcome to the CyberSource mPOS Android SDK Integration Guide. The purpose of this guide is to provide developers with a resource for understanding, integrating, and using the CyberSource Android SDK in conjunction with your CyberSource account, for payment services.

Conventions

Note, Important, and Warning Statements

Note

A *Note* contains helpful suggestions or references to material not contained in the document.

Important

An *Important* statement contains information essential to successfully completing a task or learning a concept.

Warning

A *Warning* contains information or instructions, which, if not heeded, can result in a security risk, irreversible loss of data, or significant cost in time or revenue or both.

Text and Command Conventions

Convention	Usage
Bold	▪ Field and service names in text; for example: Include the **ics_applications** field. ▪ Items that you are instructed to act upon; for example: Click **Save**.

Convention	Usage
`Screen text`	■ XML elements. ■ Code examples and samples. ■ Text that you enter in an API environment; for example: Set the **pos_cardPresent** field to `true`.

Customer Support

For support information about any CyberSource service, visit the Support Center:

http://www.cybersource.com/support

CHAPTER 1

Overview

The CyberSource Mobile Point of Sale Android SDK is a semi-integrated solution that enables you to add mobile point-of-sale functionality to your payment application, including card-present EMV capabilities (contact and contactless). The merchant's application invokes this SDK to complete an EMV transaction. The SDK handles all the complex EMV workflow as well as securely submitting the EMV transaction for processing. The merchant's application never touches any EMV or card data at any point.

Wi-Fi or cellular connectivity is required for this solution.

Android-supported device are required for integration with the CyberSource mPOS SDK.

Supported Options

Readers

Please refer to the documentation for your reader before integration the CyberSource SDK.

- BBPOS chipper 2x
- BBPOS Chipper 2x BT
- BBPOS Wisepad2*
- BBPOS Wisepad2 Plus

* BBPOS Wisepad2 is supported via BlueFin P2PE solution.

Card Brands

Contact and contactless transactions are supported by the following card brands:

- Visa
- Mastercard

- American Express
- Discover

Other card brands are accepted when processing swiped or keyed-in transactions through the card reader. Wisepad 2 and Wisepad 2 Plus accept keyed-in transactions using the terminal keypad.

Payment Types

- Authorization and Capture—Retail
- Authorization only—Endless aisle
- Auto-Authorization Reversal—When an incomplete authorization occurs, the SDK initiates authorization reversal.
- Void
- Refund
- Tokenization of card data.

Payment Acceptance

- Contact chip and signature
- Magstripe read of a chip card (fallback)
- Magstripe read of regular cards
- Keyed-in card number using the SDK (alternate solution for non-working readers)
- Keyed-in using terminal pin pad. BBPOS Wisepad2 and BBPOS Wisepad2 Plus are supported.
- All transactions from the SDK can be tokenized.

Point of Sale Flow

Below is a high-level workflow of the transaction process.

Step 1 Connect the terminal to your mobile device using either Bluetooth or the audio jack.

Step 2 Enter the amount to be charged.

Step 3 Insert or tap the EMV chip into the reader.

Step 4 The SDK will ask the shopper to Insert, Swipe or Tap their card.

Step 5 Select the application, if prompted. If there is only a compatible application on the card, the application is selected automatically.

Step 6 Confirm the amount.

Step 7 Based on the minimum amount, the merchant can skip the signature in the transaction flow.

Step 8 Remove the card when the terminal or the application prompts for card removal.

Step 9 If at any time the user cancels the transaction, the EMV transaction is canceled.

Step 10 The SDK sends an authorization request to CyberSource.

Step 11 CyberSource sends the authorization to the payment processor for card verification.

Step 12 The terminal receives the transaction result from CyberSource.

Step 13 The transaction is either approved, declined or canceled.

Transaction Flow

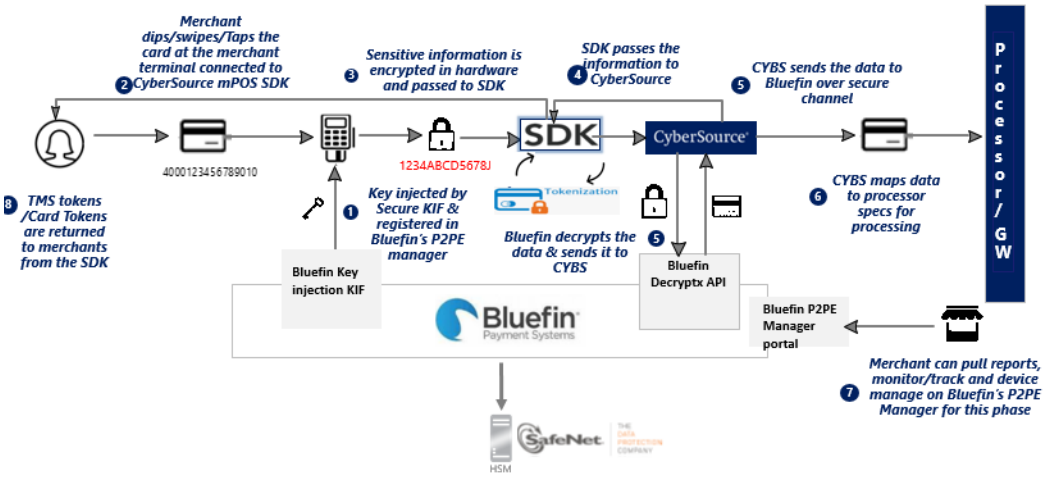
The following diagram illustrates overall solution. The following table provides a key of terminology for the diagram on the next page:

Table 1 Terminology Key

Term	Description
SDK	CyberSource mPOS SDK
P2PE	Point-to-point encryption
KIF	Key injection facility
TMS	CyberSource Token Management Services
GW	Gateway

Note

The following diagram applies to both Bluefin and Visa HSM.



1 When a customer dips, taps, swipes, or keys in a card through a Wisepad2 device, the device encrypts the card details at the hardware level, in accordance with PCI P2PE standards.

2 The CyberSource mPOS SDK connects with the reader, retrieves the encrypted payload, and submits it to CyberSource.

3 CyberSource sends the encrypted payload to either the Bluefin or Visa HSM (encryption/decryption solution), to be decrypted and parsed. The solution returns the decrypted data to CyberSource over a secure channel.

4 CyberSource sends the decrypted data and additional transaction information to your processor.

5 The SDK returns the transaction results and card data as tokens from the CyberSource platform.

Note The Visa HSM solution is supported by FutureX .

Integration Flow

The following table contains the service endpoint URLs. For testing, send requests to the Test environment, and for real payment transactions, send requests to the Production environment.

Table 2 Service Endpoints

Service	Test Environment	Production Environment
Device registration	https://authtest.ic3.com	https://auth.ic3.com

Table 2 Service Endpoints (Continued)

Service	Test Environment	Production Environment
Token creation	https://authtest.ic3.com	https://auth.ic3.com
Signature retrieval	https://mobiletest.ic3.com	https://mobile.ic3.com

Warning All request payloads should be sent in a body format other than query parameters.

Process 1—Onboard to the CyberSource Platform

Work with your CyberSource account manager to have your account created and configured on the CyberSource platform. For new or existing merchant accounts, the Mobile Point of Sale (mPOS) service should be enabled for your CyberSource account. Merchants interested in tokenization or endless-aisle tokenization services should have their account enabled for our TMS solution or secure storage products. Ask your account manager for more information.

Process 2—Create CyberSource OAuth API Keys

Based on your business needs, you can create OAuth API keys for the following use cases:

- Public—Client Credentials or Password option
- Confidential—Client Credentials or Password option

For instructions, see "Generating Client Credentials," page 15.

Process 3—Register Your Device

See "Registering the Devices," page 17.

Process 4—Generate the Authentication Token

The SDK requires an authentication token to initiate a transaction. Then **deviceId** from the registration response must be included in the request for an authentication token.

See "Generating an Authentication Token," page 20.

Process 5—Choose Your Transaction Use Cases

You can initiate a transaction using the following use cases.

Use Case 1—Retail

A retail transaction is used for the sale of goods or services. The CyberSource mPOS SDK directly connects to the payment gateway for authorization and capture of payment information.

Use Case 2—Retail Tokenization

The transaction will be authorized and tokenized automatically by connecting to our token management solution.

Use Case 3—Endless Aisle

The Endless Aisle solution enables the consumer to pay in the store and collect the goods or services at the location of their choice. CyberSource mPOS SDK calls authorization of the card for the transaction. The authorization can be captured by the merchant once the fulfillment process is completed.

Use Case 4—Endless Aisle Tokenization

The authorized card data will be tokenized by connecting to our token management solution.

Use Case 5—Tokenization

All card entry modes (contact, contactless, swipe/ MSR, or keyed- in using the card reader) will be tokenized by our Token Management Solution service. Tokens can be used to authorize and capture payment transactions.

Tokenizing the Card Data

A successful payment request returns a response that contains a payment token. The token is part of CyberSource Token Management Service. For more information about this token, see the CyberSource Documentation page for Token Management Service.

- Token use case 1—card number tokenized. Your CyberSource Technical Account Manager will configure your account to use an Instrument token.
- Token use case 2—card number, expiration date, and billing address combination tokenized. Your CyberSource Technical Account Manager will configure your account to use a Payment Instrument token.

Token use case 3—card number, expiration date, billing address, shipping address, and merchant-defined data, and consumer email address are tokenized – Your CyberSource Technical Account Manager will configure your account to a Customer token.

Use Case 6—Integrated Commerce

For an integrated commerce experience, merchants can use the same CyberSource merchant ID for E-commerce and Retail channels. The E-commerce merchant ID might be subscribed for Address Verification Services (AVS). The Retail/In-Person use-case doesn't require address verification. To provide a consistent experience across both Retail and E-commerce channels, the merchant can collect the billing and shipping address of the shopper using the SDK.

AVS validates the street address and ZIP/postal code of the shopper against the information associated with the card that was used. AVS is not supported for every processor. Contact your CyberSource account manager to configure AVS for your retail or E-Commerce accounts.

Use Case 7—Signature Retrieval

For Retail and Endless Aisle use-cases, we accept chip and signature transaction. We store the shopper’s signature that was taken during the transaction process. You can retrieve the signature from us with our API. You can also use the Skip Signature option by providing the minimum amount transaction limit in the SDK to skip signatures.

Use Case 8—ECI Indicator

Using the Electronic Commerce Indicator (ECI) value of MOTO, Mail Order / Telephone Order, your shopper can make a call to a call center to communicate their credit card number to initiate a transaction. Card numbers will be entered in via P2PE certified card reader (BBPOS – Wisepad2). With MOTO features, you can integrate your call center channel with our SDK, and enjoy unified reporting, covering both transaction flows.

Use Case 9—White Labeling

By including our SDK library in your native application, you can seamlessly transition from your UI/UX screen to the CyberSource mPOS SDK payment-processing screen. We provide fonts, background and foreground color attributes in our SDK to match your application UI/UX.

CHAPTER 2

Prerequisites

Important Before integrating the SDK into your application, you must first obtain *test* and *live* card readers, generate client credentials, register and activate the device in the CyberSource Business Center, and generate a session token as explained below.

Bluefin Solution

- BBPOS Wispad2—EMV contact, Contactless, MSR /Swipe, Keyed-in.
- BBPOS Wisepad2 is P2PE certified device with Bluefin solution.
- Reference – PCI Approval number - 4-10198

Requirements

You must have a contractual relationship with Bluefin Payment Systems for PCI-validated P2PE services, which include:

- Key injection
- Decryption, which is integrated with CyberSource
- Hardware

You can order the card readers here. Ask your CyberSource account manager for more information.

For Bluefin solutions, follow the instructions in the video guide. The video provides instructions to use the P2PE portal.

You must use a BBPOS Wisepad 2 device that is:

- provided by Bluefin Payment Systems
- injected with encryption keys for the CyberSource payment card industry (PCI) point-to-point encryption (P2PE) solution, which is powered by Bluefin

You must have separate devices for sandbox testing and production.

You must manage your Bluefin devices through the Bluefin P2PE Manager portal, which enables you to:

- Track device shipments
- Deploy, activate or terminate devices
- Manage users and administrators
- View P2PE transactions
- Download and export reports for PCI compliance

The CyberSource PCI P2PE solution, which is powered by Bluefin, does the following:

- Safeguards card data at the terminal hardware level.
- Reduces your PCI burden by minimizing the number of PCI audit questions to which you must respond.
- Provides device life cycle management through the Bluefin P2PE Manager portal.
- Supports EMV – Contact, contactless, magnetic stripe read (MSR) and manual key entry.

Generating Client Credentials

Before integrating the SDK, you must first create OAuth credentials. These credentials will be used in your API request for registering a device, creating an authentication token, and using the signature-retrieval API. Note that before you complete the following procedure, you must first contact your CyberSource account manager to configure your account for mPOS services.

To generate the client ID and client secret from the legacy Business Center:

Step 1 Log in and navigate to **Account Management > Client Integration Management**.

Step 2 Click the key icon on the left side and select **OAuth 2.0**.

Step 3 Click the **+** to create your key credentials.

Step 4 Enter the information in the Create Credentials window, shown in the Create Credentials table below.

To generate the client ID and client secret from the new Business Center:

Step 1 Log in and navigate to **Payment Configuration > Key Management**.

Step 2 Select **OAuth2.0**.

Step 3 Click the **+** to create your key credentials.

Step 4 Enter the information in the Create Credentials window, shown in the Create Credentials table below.

Table 3 Create Credentials

Type of Information	Description
Transactions Search API Permissions	Enables the user to use API transaction-search functionality.
Payments Permissions	■ Payment Credit Permission—enables the user to process credit transactions. Ability to create a refund. ■ Payment Debit Permission—enables the user to process debit transactions. Ability to create a card transaction. ■ Payment Verification Permission—enables the user to run reports, search for transactions, and view transaction details in the Business Center.
mPOS Permissions	■ mPOS Device Management—enables the user to manage devices in the Business Center's Device Management page. ■ mPOS Device Access—enables the user to access the device management API. ■ mPOS Device Terminal ID Management—enables the user to access the terminal ID of the card reader.
Configuration	■ Client Description—description of the software client. ■ Client Version—version of the software client. ■ Token Inactivity Duration—maximum time you can have the token without using it. ■ Access Token Validity Duration—maximum time for which access tokens generated by this client are valid.
Mobile Permission	Mobile endpoint access. Must be On to use the mPOS.
Client Type	■ Public—does not provide merchant with secret key. ■ Confidential— provides merchant with secret key.

Warning

Store the client credentials and secret in your server and not in your application or device.

Registering the Devices

Each *test* or *live* device that is used for transactions must be registered in its respective environment. You should set up your application to make a device registration call the first time any user logs in to your application.

- To register your device in the legacy Business Center, log in and navigate to **Account Management > Device Management** to activate your registered device.
- To register your device in the new Business Center, log in and navigate to **User Device Management > Mobile** to activate your registered device.

Enter the registration request parameters shown in the table below.

Table 4 Registration Request Parameters

Parameter Name	Description
client_id	Used only with client credentials. Unique identifier of your application on the CyberSource system. Obtain in the Business Center during key creation, as shown in the preceding section. For security reasons, you should store it in your server and not in the application.
client_secret	Used only with client credentials. Secret key associated with your application on the CyberSource system. Obtain in the Business Center during key creation, as shown in the preceding section. For security reasons, you should store it in your server and not in the application.
merchant_id	The **merchant_id** parameter contains the CyberSource merchant ID that was created when your CyberSource account was created.
device_id	Unique identifier of the device. This is a mandatory field for the creation of an authentication token. The authentication token is needed when the SDK initiates a transaction. This is a mandatory field for authentication-token creation.
description	Merchant-defined description of the application.
device_platform	Used to identify the platform on which the device runs. For example, Android 4.2.1.

Note

All request parameters should be sent in the body and not in the query.

Device Registration

The following examples show request and response when registering a device and creating keys using the public option.

Example Device Registration Request with Client Credentials

```
POST /apiauthservice/oauth/device

Content-Type: application/x-www-form-urlencoded

client_id=DigkpILFw7&client_secret=067aca61-0b75-459b-b3e8-
a0fd0a726190&merchant_id=your_merchnatid&device_
id=123456&description=Mobiletestreader&phoneNumber=5551234567&devi
ce_platform=Android&platform=1&comments=casdeviceregistermobileid1
```

Example Device Registration Request with Password

```
POST /apiauthservice/oauth/device

Content-Type: application/x-www-form-urlencoded

client_id=DigkpILFw7&username=Testuser&password=your_
password&merchant_id=your_merchnatid&device_
id=123456&description=Mobiletestreader&phoneNumber=5551234567&devi
ce_platform=Android&platform=1&comments=casdeviceregistermobileid1
```

Example Device Registration Response - Public Option

```
{    "device_id": "device1",

   "status": "pending"}
```

Device Registration Request- Confidential Option

The following examples show request and response when registering a device and creating keys using the confidential option.

Example Registration Request Using Client Credentials - Confidential

```
POST /apiauthservice/oauth/device

Content-Type: application/x-www-form-urlencoded

client_id=DigkpILFw7&client_
secret=a8d24186173d1dfc6a37c9b7f9a1daa8&merchant_id=your_merchant_
id&device_id=123456&description=Bob&phoneNumber=5551234567&device_
platform=Android&platform=1&comments=regdevicemid1
```

Example Registration Response using Client Credentials - Confidential

```
{    "device_id": “device47_location_1",    "status": "pending"}
```

Example Registration Request Using Password- Confidential

```
POST /apiauthservice/oauth/device

Content-Type: application/x-www-form-urlencoded

client_id=DigkpILFw7&client_
secret=a8d24186173d1dfc6a37c9b7f9a1daa8&merchant_id=your_merchant_
id&device_id=123456&description=Bob&phoneNumber=5551234567&device_
platform=Android&platform=1&username=cybs_username&password=cybs_
password&comments=deviceregistration
```

Example Registration Response Using Client Credentials - Confidential

```
{    "device_id": "device47_location_1 ",    "status": "pending"}
```

Activating the Device

Step 1 Log into the CyberSource Business Center.

Step 2 Navigate to **Account Management > Device Management**.

Step 3 Find the device ID of the device you would like to activate in the Device ID column.

Step 4 In the Status column, click the status for this device and select **Active**.

Step 5 To disable the device, click the status for the device and choose Disable. Disabled devices will decline the transaction.

Generating an Authentication Token

Authentication is performed by obtaining an authentication token.

To obtain a token:

Step 1 Log in to the CyberSource Business Center and navigate to **Account Management > Transaction Security Keys**.

Step 2 Click **Generate Client ID**.

Step 3 Click **Generate Secret**.

Step 4 Pass in the client ID, device ID, and the merchant credentials that you use to access the Business Center. Below is an example of a token request and response.

Requesting an Authentication Token

You can use the public option or the confidential option.

Public Option

The following examples show request and response when requesting a token using the public option.

Example Token Request with Client Credentials

```
POST /apiauthservice/oauth/token

Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&client_id=34tnl54&merchant_id=ebc_
mid&platform=1&device_id=3l5k6kn5
```

Example Token Response with Client Credentials - Public Option

```
{    "access_token": "37006d70-bf53-4941-b6d5-f719bd84845d",

  "token_type": "bearer",    "expires_in": 27916,

"scope": "mPOS_DEVICE_ACCESS

 MPOS_DEVICE_MANAGEMENT MPOS_DEVICE_TID_MANAGEMENT

 PaymentCreditPermission PaymentDebitPermission Paymen

tVerificationPermission TransactionViewPermission",

"client_status": "active"}
```

Example Token Request with Password

```
POST /apiauthservice/oauth/token

Content-Type: application/x-www-form-urlencoded
```

```
grant_type=password&client_id=DigkpILFw7&merchant_
id=username&platform=1&device_id=123456&username=ebc_
username&password=ebc_password
```

Example Token Response with Password - Public Option

```
{    "access_token": "37006d70-bf53-4941-b6d5-f719bd84845d",

  "token_type": "bearer",    "expires_in": 27916,

"scope": "MPOS_DEVICE_ACCESS

 MPOS_DEVICE_MANAGEMENT MPOS_DEVICE_TID_MANAGEMENT

 PaymentCreditPermission PaymentDebitPermission Paymen

tVerificationPermission TransactionViewPermission",

"client_status": "active"}
```

Token Request- Confidential Option

The following examples show request and response when registering a device and creating keys using the confidential option.

Example Token Creation Request Using Client Credentials - Confidential

```
POST /apiauthservice/oauth/token

Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&client_id=DigkpILFw7&client_
secret=a8d24186173d1dfc6a37c9b7f9a1daa8&merchant_id=your_merchant_
id&platform=1&device_id=123456
```

Example Token Creation Response using Client Credentials - Confidential

```
{    "access_token": "37006d70-bf53-4941-b6d5-f719bd84845d",
"token_type": "bearer",    "expires_in": 28415,    "scope": "MPOS_
DEVICE_ACCESS MPOS_DEVICE_MANAGEMENT MPOS_DEVICE_TID_MANAGEMENT
```

```
PaymentCreditPermission PaymentDebitPermission
PaymentVerificationPermission TransactionViewPermission",
"client_status": "active"}
```

Example Token Creation Request Using Password- Confidential

```
POST /apiauthservice/oauth/token

Content-Type: application/x-www-form-urlencoded

grant_type=password&client_id=DigkpILFw7&client_
secret=a8d24186173d1dfc6a37c9b7f9a1daa8&merchant_id=your_merchant_
id&device_id=123456&username=cybs_username&password=cybs_password
```

Example Token Creation Response Using Client Credentials - Confidential

```
{    "access_token": "37006d70-bf53-4941-b6d5-f719bd84845d",
"token_type": "bearer",    "expires_in": 28415,    "scope": "MPOS_
DEVICE_ACCESS MPOS_DEVICE_MANAGEMENT MPOS_DEVICE_TID_MANAGEMENT
PaymentCreditPermission PaymentDebitPermission
PaymentVerificationPermission TransactionViewPermission",
"client_status": "active"}
```

CHAPTER 3

Integrating the SDK

This section provides instructions for setting up and integrating the SDK, setting up and connecting to the terminal, selecting the decryption service, and implementing transaction use-cases.

Setting Up the SDK

Step 1 Download the *CyberSource mPOS SDK.aar* file and copy it into the *applications/lib* folder.

Step 2 In Android Studio, Navigate to **File > New > New Module**, select *import .JAR/.AAR Package*, and click **Next**.

Step 3 Provide the path of the *.aar* file and click **Finish**.

Step 4 Add this module as a dependency to your project by clicking on the project and selecting **Open Module Settings > Dependency**. Then Click **+**.

Step 5 Select the module we imported in previous step and click “OK”

Step 6 Sync the project and you are ready to use the SDK.

Instantiating the Manager

To instantiate the manager, create an instance of the **Settings** class. The **Settings** class requires parameters such as `environment`, `deviceId`, `terminalId`, `terminalAlternateId` and `merchantId`.

```
Settings settings = new Settings(Settings.Environment.ENV_
TEST,deviceID,terminalID,terminalIDAlternate,mid);

Manager manager = new Manager(settings);
```

Methods defined in **Manager** need delegate reference. This delegate is used to communicate back from the SDK to the application. The `ManagerDelegate` interface defines following methods.

```
public interface ManagerDelegate {
    public abstract void performPaymentDidFinish(PaymentResponse
paymentResponse, PaymentError paymentError);
    public abstract void refundDidFinish(PaymentResponse
paymentResponse, PaymentError paymentError);
    public abstract void voidDidFinish(PaymentResponse
paymentResponse, PaymentError paymentError);
    public abstract void
performTransactionSearchDidFinish(TransactionSearchResult
transactionSearchResult, PaymentError paymentError);
   public abstract void getTransactionDetailDidFinish(Transaction
transaction, PaymentError paymentError);
    public abstract void sendReceiptDidFinish();
}
```

Setting up and Connecting to the Terminal

The APIs for managing reader connections are part of the **ReaderConnectionManager** class. The class communicates with the merchant application through the **ReaderConnectionDelegate** interface.

Prerequisite Steps

Regardless of which connection type you use, begin with the following steps.

Step 1 Create an instance of **ReaderConnectionManager**:

```
ReaderConnectionManager mReaderConnectionManager = new
ReaderConnectionManager(context);
```

Step 2 Set the reader connection delegate to the connection manager.

```
mReaderConnectionManager.setReaderConnectionDelegate(mReaderConnec
tionDelegate)
```

Using the Audio Jack

Step 1 Set the connection type as Audio in **ReaderConnectionManager.**

```
mReaderConnectionManager.setReaderConnectionType
(EMVReaderConnectionType.AUDIO)
```

Step 2 Confirm that the reader is plugged in and call the **startAudio** API.

```
mReaderConnectionManager.startAudio();
```

The following callbacks are triggered when the device is plugged in, or is unplugged after the **startAudio** call.

```
-void onAudioDevicePlugged();

-void onAudioDeviceUnplugged();
```

To confirm that the reader connection was established successfully, call the **getDeviceInfo** API.

```
mReaderConnectionManager.getDeviceInfo(mReaderConnectionDelegate)
```

The result is returned by the **onReturnDeviceInfo ()** delegate method.

```
public void onReturnDeviceInfo(Hashtable<String, String>
deviceInfo) {

}
```

In some devices running Android OS version 7+or later, while connecting or getting device information, if you get a **COMM_ERROR** or **TIMEOUT** error, then the Audio reader needs to be calibrated using auto audio configuration.

```
public void onError(VMposError error, String s) {
       if (error == VMposErrorEMV.COMM_ERROR ||
      error == VMposErrorEMV.TIMEOUT) {
    }
 }
```

Using Bluetooth

Scanning

Step 1 Set the connection type to **Bluetooth** in **ReaderConnectionManager**.

```
mReaderConnectionManager.setReaderConnectionType
(EMVReaderConnectionType.BLUETOOTH)
```

Step 2 Use the **scanBTDevices** API of **ReaderConnectionManager** to start the scan.

```
mReaderConnectionManager.scanBTDevices(mReaderConnectionDelegate);
```

Call back method:

```
public void onBTReturnScanResults(List<BluetoothDevice>
bluetoothDeviceList) {

}
```

Connecting a Reader

Use the **connectBTDevice** API to connect a particular reader.

```
mReaderConnectionManager.connectBTDevice(bluetoothDevice,
mReaderConnectionDelegate);
```

Callback method:

```
public void onBTConnected(BluetoothDevice connectedDevice) {

}
```

Disconnecting

Use the **disconnectBTDevice** API to disconnect the reader.

```
mReaderConnectionManager.disconnectBTDevice(mReaderConnectionDeleg
ate);
```

Callback method:

```
public void onBTDisconnected(BluetoothDevice connectedDevice) {

}
```

Selecting Your Decryption Service

CyberSource provides both the Visa HSM service and the Bluefin service for decryption

- The Bluefin service is a PCI P2PE-listed solution for the Wisepad 2 terminal.

- The Visa HSM service is supported for Chipper 2x, Chipper 2x BT, Wisepad 2 and Wisepad 2 plus terminals.

Merchants interested in the Bluefin solution must procure their readers directly from Bluefin.

The **PaymentRequest** class defines the enum **PaymentRequestDecryptionServices** for Bluefin and the Visa HSM service.

```
public enum PaymentRequestDecryptionServices{
    PaymentRequestVISAHSM,
    PaymentRequestBluefin
}
```

Calling the Visa HSM Service

Use the following code to call the Visa HSM service.

```
PaymentRequest paymentReqObject = new PaymentRequest ();
paymentReqObject.setDecryptionServices(PaymentRequest.PaymentRequestDecry
ptionServices.PaymentRequestVISAHSM);
```

Calling the Bluefin Service

Use the following code to call the Bluefin service.

```
PaymentRequest paymentReqObject = new PaymentRequest ();
paymentReqObject.setDecryptionServices(PaymentRequest.PaymentReque
stDecryptionServices.PaymentRequestBluefin);
```

Payment Acceptance Mode

Transaction processing depends on which use cases you support in your application. Below are the payment acceptance modes supported by the SDK.

EMV Chip Card

An **enum** for the entry mode is defined in the **PaymentRequest** class.

```
public enum PaymentRequestEntryMode {
    PaymentRequestEntryModeSwipeOrInsertOrTap,
    PaymentRequestEntryModeSwipe,
    PaymentRequestEntryModeReaderKeyEntry,
    PaymentRequestEntryModeAppKeyEntry
}
```

Use the following methods to set and get the entry modes:

```
public PaymentRequestEntryMode getEntryMode() {
    return entryMode;
}

public void setEntryMode(PaymentRequestEntryMode entryMode) {
    this.entryMode = entryMode;
}
```

Use the following code snippet to perform an EMV chip transaction:

```
PaymentRequest paymentReqObject = new PaymentRequest();

paymentReqObject.setEntryMode(PaymentRequest.PaymentRequestEntryMo
de.PaymentRequestEntryModeSwipeOrInsertOrTap);
```


Note

When initializing the **PaymentRequest** object, enter an entry mode. Currently, we support the four entry modes shown above. If the merchant wants to use the contact method, choose **PaymentRequestEntryModeSwipeOrInsertOrTap**. This indicates that it could be an EMV transaction, swipe, contact, or contactless.

Magnetic Stripe Reader

Use the **PaymentRequestEntryModeSwipe** when initializing the **PaymentRequest** object.

```
PaymentRequest paymentReqObject = new PaymentRequest();

paymentReqObject.setEntryMode(PaymentRequest.PaymentRequestEntryMo
de.PaymentRequestEntryModeSwipe);
```

Keyed-In in Transactions Using the Terminal Keypad

Use the **PaymentRequestEntryModeReaderKeyEntry** when initializing the **PaymentRequest** object.

```
PaymentRequest paymentReqObject = new PaymentRequest();

paymentReqObject.setEntryMode(PaymentRequest.PaymentRequestEntryMo
de. PaymentRequestEntryModeReaderKeyEntry);
```

Manually Keyed-In Transactions

Entering the card number using the SDK / device is provided as a backup option when you find issues with the reader. If the store associate is outside the store with no terminal charger or a broken charger, this option comes in handy.

Use the **PaymentRequestEntryModeAppKeyEntry** when initializing the **PaymentRequest** object.

```
PaymentRequest paymentReqObject = new PaymentRequest();

paymentReqObject.setEntryMode(PaymentRequest.PaymentRequestEntryMo
de.PaymentRequestEntryModeAppKeyEntry);

VMposCardDataManual cardDataManual = new VMposCardDataManual();

String expMonth = “12”;
String expYear = “31”;
String cardNumber = “4111111111111111”;
String zip = “99508”;
String name = “Test Card”;
String cvv2 = “123”;

cardDataManual.setCardData(cardNumber, expMonth, expYear, zip,
cvv2, name);

paymentReqObject.setCardDataManual(cardDataManual);
```

Partial Authorization

Use the **setPartialAuthEnabled** method of the **PaymentRequest** class if you want the transaction to be qualified for partial authorization.

```
PaymentRequest paymentReqObject = new PaymentRequest();
paymentReqObject.setPartialAuthEnabled(true);
```

The SDK prompts the merchant to accept or decline. Based on the selection made by merchant, either a follow-on capture or an authorization reversal will be performed and the final transaction response is sent back to the application.

Processing the Transaction Results

The transaction status can be **approved**, **declined**, **canceled**, or **error**. For approved transactions, you can process the results with the following data. For more information, see "Testing and Troubleshooting," page 59.

```
public class PaymentResponse {

    public String requestID;
```

```
    public PaymentResponseDecision decision;
    public String reasonCode;
    public String requestToken;
    public String authorizationCode;
    public String authorizedDate;
    public String authorizedTime;
    public String reconciliationID;
    public PaymentResponseEmvReply paymentResponseEmvReply;
    public PaymentResponseCard paymentResponseCard;
    public HashMap<String, String> receiptValuesMap;

}

public enum PaymentResponseDecision {
    ACCEPT, ERROR, REJECT, REVIEW, UNKNOWN;
}

public class PaymentResponseEmvReply {
    public String combinedTags;
    public String decryptedRequestTags;

}

public class PaymentResponseCard {
    public String suffix;

}
```

Additional Attributes Supported in the SDK

Merchant Reference Code

We recommend you send the **merchantReferenceCode** value when initializing the **PaymentRequest** object, to benefit from linking payment requests with authorization and the merchant request.

```
PaymentRequest paymentReqObject = new PaymentRequest();

paymentReqObject.setMerchantReferenceCode(“Cybersource”);
```

Currency

The **PaymentRequest** class has member variable of type **PurchaseTotal** where you can set the desired currency. The currency is in the form of currency code. For example, USD, INR, or EUR. The default currency code is USD.

```
PaymentRequest paymentReqObject = new PaymentRequest();

PurchaseTotal purchaseTotal = new PurchaseTotal();

purchaseTotal.setCurrency(“INR”);
```

Commerce Indicator

The **VMposCommerceIndicator** enum defines various ECI indicators.

```
public enum VMposCommerceIndicator {
    VMPOS_COMMERCE_INDICATOR_RETAIL("retail"),
    VMPOS_COMMERCE_INDICATOR_INTERNET("internet"),
    VMPOS_COMMERCE_INDICATOR_RECURRING("recurring"),
    VMPOS_COMMERCE_INDICATOR_MOTO("MOTO");
}
```

Use the following code snippet to specify the commerce indicator of your choice in the payment request.

```
PaymentRequest paymentReqObject = new PaymentRequest();

paymentReqObject.setCommerceIndicator(VMposCommerceIndicator.VMPOS
_COMMERCE_INDICATOR_RETAIL);
```

Supported Services

The **PaymentRequest** class contains the enumeration parameter **supportedServices,** of type **PaymentRequestSupportedServices**.

```
public enum PaymentRequestSupportedServices {
    PaymentRequestTokenized,
    PaymentRequestEndlessAisle,
    PaymentRequestTokenizedEndlessAisle,
    PaymentRequestRetail,
    PaymentRequestTokenizedRetail
}
```

The following code snippet shows how to specify **Retail** as the supported service in the payment request.

```
PaymentRequest paymentReqObject = new PaymentRequest();

paymentReqObject.setSupportedServices(PaymentRequest.PaymentReques
tSupportedServices.PaymentRequestRetail)
```

Merchant Transaction Identifier

```
PaymentRequest paymentReqObject = new PaymentRequest();

paymentReqObject.setMerchantTransactionIdentifier(“transaction
identifier”);
```

Tokenizing the Card Data

- Token use case 1—card number tokenized. Your CyberSource Technical Account Manager will configure your account to use an Instrument token.
- Token use case 2—card number, expiration date, and billing address combination tokenized. Your CyberSource Technical Account Manager will configure your account to use a Payment Instrument token.
- Token use case 3—card number, expiration date, billing address, shipping address, and merchant-defined data tokenized – Your CyberSource Technical Account Manager will configure your account to a Customer token.

By setting supported services to **PaymentRequestTokenized**, the card data is tokenized.

```
PaymentRequest paymentReqObject = new PaymentRequest();

paymentReqObject.setSupportedServices(PaymentRequest.PaymentReques
tSupportedServices.PaymentRequestTokenized)
```

Example Using Multiple Transaction Types

The following shows settings with Secure Acceptance—the services enum is defined in the **PaymentRequest** class.

```
public enum PaymentRequestSupportedServices {
    PaymentRequestTokenized,
    PaymentRequestEndlessAisle,
    PaymentRequestTokenizedEndlessAisle,
    PaymentRequestRetail,
    PaymentRequestTokenizedRetail
}
```

Collecting Billing and Shipping Information

Billing and shipping address can be set using **BillTo** and **ShipTo** classes, which are member of **PaymentRequest** class.

```
PaymentRequest paymentReqObject = new PaymentRequest();

BillTo billTo = new BillTo(“billingFirstName”,
        “billingLastName”, “billingAddress1”,
        “billingAddress2”, “billingCity”,
        “billingState”, “billingZip”,
        “billingCountry”, “billingPhone”,
        “billingEmail”);

ShipTo shipTo = new ShipTo(“shippingFirstName”,
        “shippingLastName”, “shippingAddress1”,
        “shippingAddress2”, “shippingCity”,
        “shippingState”, “shippingZip”,
        “shippingCountry”, “shippingPhone”,
        “shippingEmail”);
```

```
paymentReqObject.setBillTo(billTo);
paymentReqObject.setShipTo(shipTo);
```

Adding Merchant-Defined Data

There is a property defined in the **PaymentRequest** class (**merchantDefinedDataList**), which is an array that enables the merchant to pass values along with the transaction request. When setting the value, initialize the request object, and then assign the property value:

```
ArrayList<String> merchantDefinedDataFields = new ArrayList<>();
merchantDefinedDataFields.add(“Field 1”);
merchantDefinedDataFields.add("Field 2");
merchantDefinedDataFields.add("Field 3");
merchantDefinedDataFields.add("Field 4");
merchantDefinedDataFields.add("Field 5");

PaymentRequest paymentReqObject = new PaymentRequest();

paymentReqObject.setMerchantDefinedDataList(merchantDefinedDataFie
lds)
```

Generating Your Own Receipt

After the transaction, a payment response object is returned in the following callback method:

```
public void performPaymentDidFinish(PaymentResponse
paymentResponse, PaymentError paymentError) {

}
```

Using the **PaymentResponse** object, you can generate your own receipt page. While doing so you can also let the SDK know not to display the receipt dialog by using the following code:

```
PaymentRequest paymentReqObject = new PaymentRequest();

paymentReqObject.setShowReceipt(false);
```

Adding Tax, Line Items, and MCC Details to the SDK

Use **VMposItem** to pass the line items to the transaction request.

```
ArrayList<VMposItem> items = new ArrayList<VMposItem>();

BigDecimal itemPrice  = new BigDecimal("11.20”);

BigDecimal itemTaxAmount  = new BigDecimal("1.20”);

VMposItem item = new VMposItem("Item name", itemPrice,
itemTaxAmount, 1, VMposProductCodes.VMPOS_PRODUCT_CODE_HANDLING_
ONLY, "4gduitt5222");

items.add(item);

PaymentRequest paymentReqObject = new PaymentRequest();

paymentReqObject.setPosItems(items);
```

Use **MerchantDescriptor** class to provide MCC details which can be used to show store information on the paper print receipts.

```
MerchantDescriptor merchantDescriptor = new MerchantDescriptor();
merchantDescriptor.setBusinessName("CyberSource mPOS Store");
merchantDescriptor.setStreet1("901 Metro center Blvd");
merchantDescriptor.setCity("Foster City");
merchantDescriptor.setState("CA");
merchantDescriptor.setPostalCode("94404");
merchantDescriptor.setPhone("650-302-7012");
merchantDescriptor.setEmail("usriniva@visa.com");
```

```
Manager.setMerchantDescriptor(merchantDescriptor);
```

Fallback

When the chip card is damaged, the terminal will ask the consumer to swipe the card. This use case is called fallback.

When the transaction starts, the following message is displayed by the SDK: “Please swipe, insert, or tap card”.

When a damaged or incorrect card inserted in the terminal, the following SDK message is displayed: “NOT ICC CARD FALLBACK”.

The application displays the following message: “Non-Chip Card, please swipe.”

Searching for a Payment

When the transaction is complete, you can search for the transaction by integrating the **searchpayment** API using the SDK.

The **Manager** class exposes an API to perform transaction search. The API signature is:

```
public void performTransactionSearch(TransactionSearchQuery
searchQuery, String accessToken, final ManagerDelegate
managerDelegate)
```

- **searchQuery**: Object of type **TransactionSearchQuery.**
- **accessToken**: OAuth access token.
- **managerDelegate**: Callback interface of type **ManagerDelegate.**

The **TransactionSearchQuery** object serves the purpose of search criteria. It has defined various instance variables that can be set to specify those search criteria.

1 Filter Type: filter by one of the following.

```
public enum TransactionSearchQueryFilter{
      AccountSuffix,
      AccountPrefix,
      AccountPrefixAndSuffix,
```

```
        LastName,
        DeviceId,
        MerchantReferenceCode
}
```

2 Account Suffix: Last 4 digits of account number.

3 Account Prefix: First 4 digits of account number.

4 Last Name: Last name of merchant.

5 Device Id: Device ID.

6 Merchant Reference Code: Reference code used for transactions.

7 Start Date: Start date of the date range.

8 End Date: End date of the date range

Assuming a UI screen with the above input text fields, the following code snippet demonstrates how to create the **TransactionSearchQuery** object and call the **performTransactionSearch()** API.

Creating the Search Query Object

```
TransactionSearchQuery searchQuery = new TransactionSearchQuery();
if(first4.getText().length() > 0 && last4.getText().length() > 0){

searchQuery.setFilter(TransactionSearchQuery.TransactionSearchQuer
yFilter.AccountPrefixAndSuffix);
    searchQuery.setAccountPrefix(first4.getText().toString());
    searchQuery.setAccountSuffix(last4.getText().toString());
}else if(merchantReferenceNo.getText().length() > 0){

searchQuery.setFilter(TransactionSearchQuery.TransactionSearchQuer
yFilter.MerchantReferenceCode);
searchQuery.setMerchantReferenceCode(merchantReferenceNo.getText()
.toString());

}else if(lastName.getText().length() > 0){

searchQuery.setFilter(TransactionSearchQuery.TransactionSearchQuer
yFilter.LastName);
searchQuery.setLastName(lastName.getText().toString());

}else if(deviceId.getText().length() > 0){

searchQuery.setFilter(TransactionSearchQuery.TransactionSearchQuer
```

```
yFilter.DeviceId);
searchQuery.setDeviceId(deviceId.getText().toString());

}

long dateFromValue= 0, dateToValue=0;
if(dateFrom.getText().length() > 0){

dateFromValue =
Utils.stringDateToMillis(dateFrom.getText().toString()+"
00:01:00");

if(dateFromValue > 0){

     searchQuery.setDateFrom(dateFromValue);
}

}


if(dateTo.getText().length() > 0){

dateToValue =
Utils.stringDateToMillis(dateTo.getText().toString()+" 23:59:00");

    if(dateToValue > 0){
        searchQuery.setDateTo(dateToValue);
    }
}
```


Note

The **Utils.stringDateToMillis** utility method is converting the date string to its corresponding **Date.getTime()** long value.

Calling the Manager API

```
manager.performTransactionSearch(searchQuery, oAuthToken,
managerDelegate);
```

Retrieving the Search Result

The search result will be sent back to the application via the delegate method **performTransactionSearchDidFinish()**.

```
@Override
public void
performTransactionSearchDidFinish(TransactionSearchResult
transactionSearchResult, PaymentError paymentError){

}
```

Voiding a Transaction

```
VoidRequest voidRequestObject = new VoidRequest();
voidRequestObject.setMerchantID(merchantID);
voidRequestObject.setAccessToken(oAuthToken);
voidRequestObject.setCurrency(“USD”);
voidRequestObject.setAmount(amount);
voidRequestObject.setMerchantReferenceCode(“merchant
ReferenceCode”);
voidRequestObject.setTransactionID(transationID);

manager.performVoid(voidRequestObject,managerDelegate);
```

Void transaction result will be returned through the following callback method:

```
@Override
public void voidDidFinish(PaymentResponse paymentResponse,
PaymentError paymentError){

}
```

Refund a Transaction

```
RefundRequest refundReqObject = new RefundRequest();
refundReqObject.setMerchantID(merchantID);
refundReqObject.setAccessToken(oAuthToken);
refundReqObject.setCurrency(“USD”);
refundReqObject.setAmount(amount);
refundReqObject.setMerchantReferenceCode(“merchantReferenceCode”);
refundReqObject.setTransactionID(transactionID);

manager.performRefund(refundReqObject,managerDelegate);
```

Refund transaction result will be returned through following callback method:

```
@Override
public void refundDidFinish(PaymentResponse paymentResponse,
PaymentError paymentError){

}
```

Partial Refund

To process a partial refund transaction, navigate to the History screen and search for transactions. Select a transaction to view its details. In the Transaction Details screen, select the option to do a partial refund. Enter the amount you want to refund and click **OK**. The amount is refunded and transaction details are updated.

Capturing an Authorization for the Endless Aisle Use Case

Will be available in early FY 2019.

Retrieving a Signature

You can retrieve the signature stored against a particular transaction. Use the following **getSignature()** API from the **Manager** class

```
public void getSignature(String transactionId, String accessToken,
final VMposTransactionSignatureDelegate signatureDelegate){
```

```
}
```

- **transactionID**—The transaction ID of the signature being retrieved.
- **oAuthToken**—OAuth access token.
- **signatureDelegate**—Callback interface of type **VMposTransactionSignatureDelegate.**

```
VMposTransactionSignatureDelegate signatureDelegate = new
VMposTransactionSignatureDelegate() {
    @Override
    public void onErrorReceived(VMposError error) {

    }

    @Override
    public void
onGetSignatureFinished(VMposCyberSourceSignatureManager.VMposSigna
tureResponse  response) {

    }
};
```

If signature retrieval is successful, the **onGetSignatureFinished** callback is invoked with **VMposCyberSourceSignatureManager**.**VMposSignatureResponse** response object. This response object contains signature image stored as byte array, which can be converted to Bitmap as shown below:

```
final Bitmap signatureBitmap =
BitmapFactory.decodeByteArray(response.getImageData(), 0,
response.getImageData().length);
```

Skipping the Signature

The **PaymentRequest** class has a boolean member named **isSignatureRequired**. Depending on the requirement, we can set and reset it to control the signature screen.

```
public boolean isSignatureRequired() {
    return isSignatureRequired;
}

public void setSignatureRequired(boolean signatureRequired) {
    isSignatureRequired = signatureRequired;
}
```

After initializing the **PaymentRequest** object, set **isSignatureRequired** to false to skip the signature panel shown at the end of the transaction processing flow, which comes from the SDK. Set **true** to show signature. The default value is **true**. You can apply your own logic to hide or show signature. For example, if the amount is greater than or equal to 10, set true, otherwise false.

```
PaymentRequest paymentReqObject = new PaymentRequest();

paymentReqObject.setSignatureRequired(isSignatureRequired());
```

The definition of the **isSignatureRequired()** method can be defined as follows:

```
private boolean isSignatureRequired(){
    boolean signatureRequired = false;

if(paymentReqObject.getPurchaseTotal().getGrandTotalAmount().compa
reTo(getSignatureAmount()) > 0){
        signatureRequired = true;
    }

    return  signatureRequired;
}



public BigDecimal getSignatureAmount() {
    String storedAmount = sharedPreferences.getString("pref_key_
sign_amount", "0.00");
    if(storedAmount != null && !storedAmount.isEmpty()){
        return new BigDecimal(storedAmount);
    }
```

```
    return new BigDecimal("0.00");
}
```

Overriding the Terminal ID and Alternate Terminal ID

The **Manager** object takes instance of **Settings** class.

```
public class Settings {
    public enum Environment {ENV_TEST, ENV_LIVE, ENV_STAGE};
    private Environment environment;
    private String deviceID;
    private String terminalID;
    private String terminalIDAlternate;
    private String mid;
    private String cardMode;
}
```

Create an instance of Settings class:

```
Settings settings = new Settings();

settings.setTerminalID(“terminalID1234”);

settings.setMid(“mid1234”);
```

Use the above settings object to create instance of Manager class:

```
Manager manager = new Manager(settings);

manager.performPayment(paymentRequesObject, this,
managerDelegate);
```

Start a transaction with this **Manager** instance, your merchant ID (**MID**), **terminalIDAlternate,** and **terminalID** will be part of the payment request.

Designing a User Interface

To customize your own transaction-processing page, you can initialize the **UISettings** object and set its values. Following are the list of properties that you can use to customize the look and feel of the SDK screens.

- **backgroundColor** – SDK screen's background color;
- **spinnerColor** – Progress dialog spinner color;
- **textLabelColor** – Text color for labels in SDK;
- **detailLabelColor** – Text label color used in receipt screen;
- **textFieldColor** – Input Text Field color for keyed-in screen;
- **placeHolderColor** – Hint text color for keyed-in screen;
- **signatureColor** - Signature color;
- **fontFamily** – Font TypeFace;
- **fontStyle** – Font Style;
- **titleImage** – Image Bitmap;
- **titleImageURL** – Image URL that will get downloaded automatically by SDK;
- **titleImageResource** – Image resource ID;
- **headerTextSize** – Text size of header;
- **level1TextSize** – Size of text labels;
- **activityVerticalMargin** – Vertical margin of SDK screen;
- **activityHorizontalMargin** – Horizontal margin of SDK screen;
- **buttonHeight** – Height of the buttons;

All the colors are hex codes.

```
public class UISettings {
    private String backgroundColor;
    private String spinnerColor;
    private String textLabelColor;
    private String detailLabelColor;
    private String textFieldColor;
    private String placeHolderColor;
```

```
    private String signatureColor;
    private Typeface fontFamily;
    private int fontStyle;
    private Bitmap titleImage;
    private String titleImageURL;
    private int titleImageResource;
    private float headerTextSize;
    private float level1TextSize;
    private int activityVerticalMargin;
    private int activityHorizontalMargin;
    private int buttonHeight;
}
```

Use the setters methods of any property that you want to customize.

Registering a Device Using the Sample Application

To register a device using the sample application:

Step 1 Open the sample application and select **Register Device** from the application menu.

Step 2 When prompted, enter your credentials.

Step 3 When prompted, enter the ID of the device being registered. If you do not enter an ID, the application creates one.

Step 4 Click **Register Device**. The device status is shown as Pending.

Step 5 Log in to the CyberSource Business Center and navigate to Account Management > Device Management > Mobile Devices.

Step 6 Select the device and click **Activate**.

Setting Up Multiple User Accounts Using a CSV File

You can set up multiple user accounts at one time by adding them to a CSV file. This comes in handy when you have multiple stores and you want to set up separate user accounts for each of them. Use the following procedure.

To set up multiple user accounts:

Step 1 Create a .csv file with the following columns:

- Store_Name
- CyberSource_MID
- Device_ID
- Client_id
- Terminal_ID

Step 2 Add the information for each store and transfer the file to a mobile device that has the sample application installed.

Step 3 Open the sample application and navigate to the **Settings** page.

Step 4 Click **Stores**.

Step 5 Click **Select Your File** and select the .csv file.

Step 6 Click **Create Store**. The stores are created and you can select a store from the Store drop-down menu in the Settings page.

Over-the-Air Updates

Due to mandates or other conditions we might request that the merchant update their firmware from time to time, and other configuration updates become available as well. These updates will be provided over the air to the terminal in an OTA update. If only a firmware update is required, without the full OTA update, see Step 4 of OTA Update Sample Code.

The following must first be considered:

- Make sure you connect to a reader either through Bluetooth or the audiojack.
- Choose Demo or Production mode.

The SDK provides the **OTAUpdateManager** class with all the required methods to perform OTA updates.

```
public OTAUpdateManager(Context context, boolean demoMode,
ReaderConnectionDelegate connectionDelegate)
```

- **context**: The application/activity context.
- **demoMode** : **true** if it is a test reader, **false** if it's a production reader.
- **ReaderConnectionDelegate**: Instance of delegate object.

The **ReaderConnectionDelegate** is an interface that has callback methods to communicate back to the application about reader's connectivity progress.

```
public interface ReaderConnectionDelegate {
    void onBTReturnScanResults(List<BluetoothDevice> list);

    void onBTScanTimeout();

    void onBTScanStopped();

    void onBTRequestPairing();

    void onBTConnected(BluetoothDevice bluetoothDevice);

    void onBTDisconnected();

    void onUsbConnected();

    void onUsbDisconnected();

    void onSerialConnected();

    void onSerialDisconnected();

    void onReturnDeviceInfo(Hashtable<String, String> hashTable);

    void onBatteryLow(BBDeviceController.BatteryStatus
batteryStatus);

    void onAudioDevicePlugged();

    void onAudioDeviceUnplugged();

    void onError(VMposError error, String s);

    void onNoAudioDeviceDetected();

    void onDeviceHere(boolean b);

    void onPowerDown();

    void onPowerButtonPressed();

    void onDeviceReset();
```

```
    void onEnterStandbyMode();

    void onBarcodeReaderConnected();

    void onBarcodeReaderDisconnected();

}
```

The **OTAUpdateManager** needs a reference of the **OTAUpdateListener** interface to communicate back to the application about the OTA checks and progress:

```
public interface OTAUpdateListener {
    void onReturnOTAUpdateProgress(OTAUpdateStatus updateStatus,
double progress);
    void
onReturnCheckForUpdateResult(OTAUpdateManager.OTACheckResult
checkResult);
    void onReturnOTAUpdateResult(OTAUpdateType updateType,
OTAUpdateResult result, String description);
    void onReturnDeviceInfo(Hashtable<String, String> deviceInfo);
    void onReturnOTAUpdateError(OTAUpdateResult result, String
description);
    void onReturnBBPOSError(OTAUpdateError result, String
description);
}
```

OTA Update Workflow

Step 1 Connect the reader.

Step 2 Check for update.

Step 3 Perform desired updates (Firmware/Configuration/ALL).

Step 4 Finish update.

The **OTAUpdateManager** class provides following methods:

```
public void checkForUpdates()
```

```
public void startOTAUpdate(OTAUpdateType updateType, boolean
demoMode, OTAUpdateListener listener)
```

- **updateType**: Type of update we want to perform
- **demoMode**: **true** if it's a test reader, **false** if it's a production reader
- **listener**: Instance of **OTAUpdateListener**

OTA Update Sample Code

Create an instance of **OTAUpdateManager** class and set **OTAUpdateListener** to it.

```
OTAUpdateManager  mOTAOtaUpdateManager = new
OTAUpdateManager(context, true, mReaderConnectionDelegate);

mOTAOtaUpdateManager.setOtaUpdateListener(otaUpdateListener);
```

Connect the reader

To connect the reader first initiate Bluetooth scan:

```
mReaderConnectionManager.scanBTDevices(mReaderConnectionDelegate);
```

The **onReturnBluetoothDevices()** callback method will receive set of Bluetooth devices found in scan.

```
@Override
public void onBTReturnScanResults(List<BluetoothDevice>
bluetoothDeviceList) {


}
```

From the above list, select the desired Bluetooth reader and call **connectBluetoothDevice()** method passing **Context**, demo mode as **true**, **OTAUpdateListener** callback and the selected Bluetooth device:

```
mReaderConnectionManager.connectBTDevice(bluetoothDevice,
mReaderConnectionDelegate);
```

If the connection has been made successfully **onBluetoothDeviceConnected()** callback method will be invoked by SDK.

```
@Override
public void onBTConnected(BluetoothDevice connectedDevice) {

}
```

Check for Update

Once reader is connected, we can check if any update to the reader is available or not by calling **checkForUpdates ()** method.

```
mOTAOtaUpdateManager.checkForUpdates();
```

The above call will connect to terminal management system and check against the connected reader for any update availability and the result will be returned in **onReturnCheckForUpdateResult** callback method.

The result parameter **OTAUpdateManager.OTACheckResult** has helper methods to find out if any update is required or not.

```
@Override
public void
onReturnCheckForUpdateResult(OTAUpdateManager.OTACheckResult
checkResult){
    if (checkResult != null) {
        if (checkResult.isUpdateRequired()) {
        UPDATE REQUIRED
        } else {
        NO UPDATE REQUIRED
        }
    }
}
```

The **OTACheckResult** class has helper methods **getRequiredUpdateType()** to let you know the type of update available. These update types are defined in the **OTAUpdateType** enum.

```
public enum OTAUpdateType {
    FIRMWARE,
    CONFIG,
    FIRMWARE_AND_CONFIG,
    NONE;
}
```

Perform Update

If update is required, call **startOTAUpdate()** method passing the desired update type. The following code snippet will update both firmware and configuration.

```
mOTAOtaUpdateManager.startOTAUpdate(OTAUpdateType.FIRMWARE_AND_
CONFIG, isDemoMode, otaUpdateListener);
```

Progress of update will be notified through **onReturnOTAUpdateProgress()** callback.

```
@Override
public void onReturnOTAUpdateProgress(OTAUpdateStatus
updateStatus, double progress) {

}
```

The **updateStatus** is an enum which tells the status of update and progress will tell the progress percentage. Possible values for **OTAUpdateStatus** are:

```
public enum OTAUpdateStatus {
    UPDATE_REQUESTED,
    UPDATING_CONFIG,
    UPDATING_FIRMWARE,
```

```
    WAITING_FOR_DEVICE
}
```

The outcome of OTA update is notified through **onReturnOTAUpdateResult** callback method.

```
@Override
public void onReturnOTAUpdateResult(OTAUpdateType updateType,
OTAUpdateResult result, String description) {

}
```

Values defined in **OTAUpdateResult** are:

```
public enum OTAUpdateResult {
    SUCCESS(0, "OTA operation is successful."),
    BATTERY_LOW_ERROR(1, "Battery level at 50% or more for Remote
updates."),
    SETUP_ERROR(2, "OTA setup error."),
    DEVICE_COMM_ERROR(3, "Device communication error."),
    SERVER_COMM_ERROR(4, "Server communication error"),
    FAILED(5, "OTA operation failed."),
    STOPPED(6, "Remote update stopped."),
    NO_UPDATE_REQUIRED(7, "No update available."),
   INCOMPATIBLE_FIRMWARE_HEX(8, "Input firmware hex not compatible
with the device."),
    INCOMPATIBLE_CONFIG_HEX(9, "Input config hex not compatible
with the device."),
    UNKNOWN(10, "Unknown error."),
    UPDATE_TYPE_NOT_SUPPORTED(11,"Update type not supported for
this reader");
}
```

If the result is SUCCESS, it implies that OTA updates has successfully completed and we can move to the final step.

Finish update

As the last step, we should make sure that reader hardware and SDK are in correct state. To do so, call the following method.

```
mOTAOtaUpdateManager.finishOTAUpdate();
```

After the reader has updated successfully, it takes around 45 – 60 seconds for reboot. Please wait for the reader to boot-up completely and connect to it again before executing next set of commands.

Setting Reader Connection Type

The default connection type in the SDK is Bluetooth. The **ReaderConnectionManager** class has method to change the connectivity type of the reader (Audio/Bluetooth).

```
public void setReaderConnectionType(EMVReaderConnectionType
connectionType) {


}
```

Where `connectionType` is an enum defined as -

```
public enum EMVReaderConnectionType {
    AUDIO,
    BLUETOOTH
}
```

If you use an audio jack reader, you can use the following method to change the connection type.

```
mOTAOtaUpdateManager.setReaderConnectionType(EMVReaderConnectionTy
pe.AUDIO);
```

To set it back to Bluetooth mode, use:

```
mOTAOtaUpdateManager.setReaderConnectionType(EMVReaderConnectionTy
pe.BLUETOOTH);
```

Sending Email Receipt

The Manager class provides sendReceipt() API using which an email receipt can be sent.

```
public void sendReceipt(ReceiptRequest receiptRequest, final
ManagerDelegate managerDelegate){

}
```

- **receiptRequest** : Object of type **ReceiptRequest**
- **managerDelegate**: Object of type **ManagerDelegate** to receive callbacks

The **ReceiptRequest** class defines following member fields that can be set using their respective setter methods.

```
public class ReceiptRequest {
    private String transactionID;
    private String toEmail;
    private String fromEmail;
    private String emailSubject;
    private String merchantDescriptor;
    private String merchantDescriptorStreet;
    private String merchantDescriptorCity;
    private String merchantDescriptorState;
    private String merchantDescriptorPostalCode;
    private String merchantDescriptorCountry;
    private String merchantReferenceCode;
    private String authCode;
    private String shippingAmount;
    private String taxAmount;
    private String totalPurchaseAmount;
    private String accessToken;
```

```
        private VMposCurrency vMposCurrency;
        private List<VMposItem> items;
        public HashMap<String, String> receiptTagValueMap;
    }
```

The following code snippet shows an example of how you can use **PaymentResponse** and **PaymentRequest** objects of a particular transaction to create a **ReceiptRequest** instance.

```
private void createReceiptRequestObject(){
    ReceiptRequest receiptRequestObject = new ReceiptRequest();
    PaymentRequest paymentRequest =
PaymentUtil.getInstance().getPaymentRequest();
    PaymentResponse paymentResponse =
PaymentUtil.getInstance().getPaymentResponse();
    String oAuthToken = PaymentUtil.getInstance().getoAuthToken();
    if(paymentResponse != null){

receiptRequestObject.setTransactionID(paymentResponse.getRequestID
());

receiptRequestObject.setAuthCode(paymentResponse.getAuthorizationC
ode());

receiptRequestObject.setReceiptTagValueMap(paymentResponse.getRece
iptValuesMap());
    }
    if(paymentRequest != null){

receiptRequestObject.setMerchantReferenceCode(paymentRequest.getMe
rchantReferenceCode());

receiptRequestObject.setTotalPurchaseAmount(paymentRequest.getPurc
haseTotal().getGrandTotalAmount() + "");

receiptRequestObject.setItems(paymentRequest.getPosItems());
    }
    receiptRequestObject.setToEmail(toEmail);
    receiptRequestObject.setFromEmail("no-reply@cybersource.com");
    receiptRequestObject.setEmailSubject("Your Transaction
Receipt");
    receiptRequestObject.setMerchantDescriptor("Cybersource");
    receiptRequestObject.setMerchantDescriptorStreet("P.O. Box
8999");
    receiptRequestObject.setMerchantDescriptorCity("San
Francisco");
    receiptRequestObject.setMerchantDescriptorState("CA");
    receiptRequestObject.setMerchantDescriptorPostalCode("94128-
```

```
8999");
    receiptRequestObject.setMerchantDescriptorCountry("USA");
    receiptRequestObject.setShippingAmount("0.0");
    receiptRequestObject.setTaxAmount("0.0");
    receiptRequestObject.setAccessToken(oAuthToken);
    receiptRequestObject.setCurrency(transactionCurrency);
}
```

CHAPTER 4

Testing and Troubleshooting

100 is a successful response code. Decline codes are listed below.

Table 5 Decline

Decline Code	Message Status	Message Displayed in the SDK
100	Successful transaction.	SUCCESS
101	The Request is missing one or more required fields.	DECLINED
102	One or more fields in the request contains invalid data.	DECLINED
110	Partial Transaction.	SUCCESS
150	General System failure.	DECLINED
200	The authorization request was approved by the issuing bank, but declined by CyberSource because it did not pass the Address Verification System (AVS) check.	DECLINED
201	The issuing bank has questions about the request.	DECLINED
202	Expired card.	DECLINED
204	Insufficient funds in the account.	DECLINED
207	Issuing bank unavailable.	DECLINED
208	Inactive card or card not authorized for card not present transactions.	DECLINED
208	CVN did not match.	DECLINED
211	Invalid CVN.	DECLINED

SDK Error Messages

The following tables show the error message for various situations, and solutions for fixing them.

Table 6 SDK Authentication

Use Case	SDK Error Message	Details	How To Fix
The device ID is not registered.	Invalid Device	The device ID that was entered is not available for the client.	Verify that the device ID sent in the request is valid.
The login credentials failed.	Invalid Grant	The merchant username and password is wrong.	Possible solutions are 1. Ensure that the username and password details are correct. 2. Ensure that the Merchant ID is correct.
The device is inactive.	Invalid Device	The status of the registered device is Inactive.	Ensure that the registered device status is active for the client.
The token is not valid	Invalid Token	The **access** token is not valid.	Generate a new **access** token and try again.
The token has expired.	Invalid Token	The **access** token is expired.	Generate a new **access** token and try again.

Table 7 Terminal Error Messages

Use Case	SDK Error message	Details	How To Fix
The is a communication problem between the devices.	Device Communication Error	A communication error occurred between the Android OS device and the mPOS device.	Possible causes include: 1. The reader's battery is missing or not charged. 2. The audio jack is defective. 3. There is o response from the reader, or the device is defective.
The reader is busy.S	Device Busy	The reader is busy performing an operation.	Restart the reader and try again.
The audio reader permission is denied.	Audio Permission denied.	Audio reader permission is denied when microphone access is not allowed.	Verify that microphone access is allowed for the mPOS application.
The Bluetooth reader is not connected.	No BLE support	Bluetooth readers will not connect to devices which do not have BLE support.	Use a BLE-supported device for Bluetooth readers.
The Bluetooth reader pairing was unsuccesful.	Bluetooth already connected	Bluetooth already connected to another device.	Disconnect the Bluetooth reader from the other paired device and try again.

Table 7 Terminal Error Messages (Continued)

Use Case	SDK Error message	Details	How To Fix
The check for OTA updates failed	Remote update requires battery level at 50% or above	OTA operations will not work if the reader battery is less than 50%.	Charge the reader to above 50%.
The check for updates failed.	OTA Operation Failed	OTA operations will not work if the reader is not registered in the TMS portal.	Verify that the reader is registered and active in TMS portal. Otherwise, contact the TMS support team.
The input firmware is not compatible.	Input firmware hex not compatible with the device.	The terminal is not compatible with the available firmware upgrade.	Contact the TMS support team with terminal details.
The input firmware configuration is not compatible with the device.	Input firmware config not compatible with the device.	The available input firmware configuration is not compatible with your current device.	Contact the TMS support team with the terminal details.
The OTA update failed.	Server communication error	When the TMS portal or APIs are not working, a server error occurrs.	Verify that the TMS Portal or APIs are working.

Table 8 Payment Processing Error Messages

Use Case	SDK Error Message	Details	How To Fix
The transaction timed out.	Transaction Timed out	The transaction can time out when waiting for the card, requesting the amount, terminal time, or confirmation from the application.	Retry the transaction with swipe or EMV mode.
EMV transaction selected when using a non-EMV card.	A non-chip card is inserted	The card inserted is not a chip card.	Use swipe mode for successful transactions when a Magnetic Stripe Reader-only card.
The selected decryption service is invalid for the reader.	Transaction Declined	The transaction is declined with “unable to decrypt payment data” error when the selected decryption service is not valid for the reader.	Select the appropriate decryption service (VISA HSM or Bluefin), depending on the reader being used, and retry the transaction.
The transaction was canceled or timed out during checkout.	Transaction cancelled or timed out	The transaction was canceled or timed out while waiting for the card or requesting amount/ terminal time from the application.	Restart the application and reader and retry the transaction.

Table 8 Payment Processing Error Messages (Continued)

Use Case	SDK Error Message	Details	How To Fix
Battery too low for transaction.	Battery is critically low	When the reader battery is critically low it cannot process the transaction.	Recharge the reader and retry the transaction.

Troubleshooting Steps

Bluetooth Terminal/Reader Connection Issues

If the reader is not detected by the SDK, try the following steps:

Step 1 Using our demo application, navigate to the Reader Setup & OTA section.

Step 2 Click **Connect BT Reader**. It will display a list of nearby Bluetooth readers.

Step 3 Select the terminal/reader you intend to use.

Step 4 When the reader connects successfully, a message with the reader's name is shown. In addition, the Reader Status will be updated to Connected.

Audio Terminal/Reader Connection Issues

If the reader is not detected by the SDK, try the following steps:

Step 1 Using our demo application navigate, to the Reader Setup & OTA section.

Step 2 Toggle the reader type switch to 0Audio.

Step 3 Click **Start Audio Reader**. This will initiate the audio reader.

Step 4 To verify the connectivity, use Get Device Information. If the connection was established successfully, it will show up reader's information dialog.

Declined Transaction—Request ID Not Generated

If the reader is connected and the transaction is declined in the terminal, the request has not reached the payment gateway. Try the following:

- Check the Device ID, merchant ID, and authorization token created using OAuth credentials. The registered device ID should be active for successful SDK authentication-token creation.

- If you are using WiFi, check your network connectivity on the access URL.
- Use the Keyed-In option via the SDK to eliminate the card reader component in the flow for debugging issues.
- Use the demo app, enter the Merchant ID, OAuth credentials, and Device ID (case sensitive) to run a transaction.

Declined Transaction—Internal Error

Contact the CyberSource Support team and provide them with the request ID that was generated in the response.

Declined Transaction—203 Error

If the SDK provides request ID with the decline 203, the card has been rejected by the processor. Try a different card.

Tokenization Failed in Card-Present Mode

For a subscription failure, check if the AVS verification check has rejected the transaction. AVS checks can be turned off by our support team. To pass the address verification check, the billing address of the card holder should be passed in the SDK.

Offline Decline

If the SDK authentication succeeded, but the terminal request is not able to reach the CyberSource end-point (for example, WiFi connectivity was lost), the terminal will decline the transaction as an “offline decline”. For offline declines, the terminal will process the EMV tags and the user will see the EMV tag response in the receipt page.

Brand Certification

Brand certified with FDI global. For other processors, contact the CyberSource Support team.