CyberSource Mobile Point of Sale

iOS 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

Contents

Recent Revisions to This Document

Release	Changes
April 2019	■ Updated the code sample in "Processing the Transaction Results," page 38. ■ Added "Partial Authorization Transactions," page 37. ■ Added "Partially Refunding a Transaction," page 53. ■ Added "Registering the Device from the Sample Application," page 58. ■ Added "Setting Up Multiple User Accounts," page 59.
January 2019	This is the first release of this document.

About This Guide

Welcome to the CyberSource mPOS iOS SDK Integration Guide. The purpose of this guide is to provide developers with a resource for understanding, integrating, and using the CyberSource iOS 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 iOS 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.

iOS-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

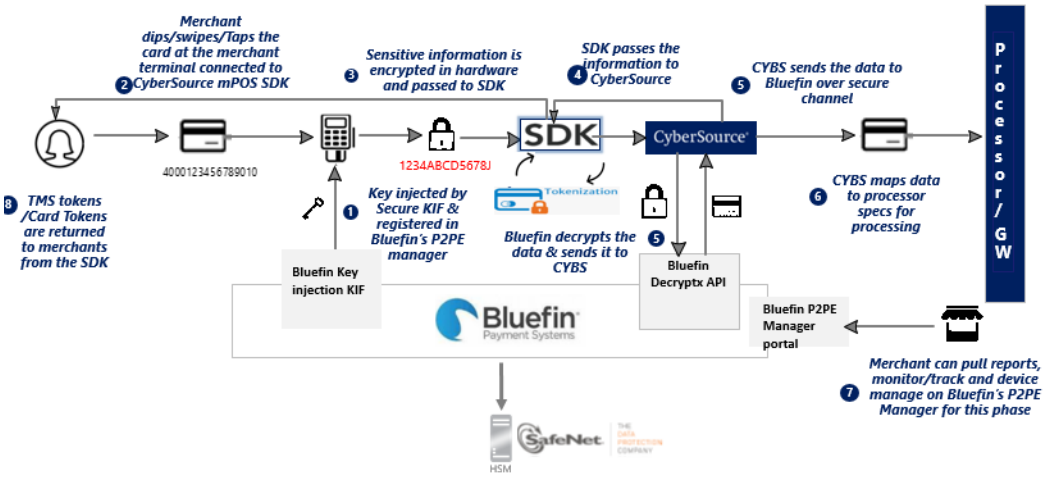
The following diagram illustrates overall solution. The following table provides a key of terminology for the diagram on the next page:

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 18.

Process 3—Register Your Device

See "Registering the Devices," page 20.

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 23.

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=iOS&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=iOS&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=iOS&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=ios&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 Include **CYBSMposKit.framework** in the application. Select **Target, In Embedded Binaries**, click the plus (**+**) and select the framework.

Once included, confirm that the path to these frameworks is added correctly in **Search Paths > Build Settings**.

Step 2 If the application is developed in the Swift language, the application must have a bridging header file created because the **CYBSMposKit.framework** is based on the Objective C language.

Example CybsMposDemo-Bridging-Header.h

```
#ifndef CYBSMposDemo_Bridging_Header_h
#define CYBSMposDemo_Bridging_Header_h
#import <CYBSMposKit/CYBSMposKit.h>
#endif
```

Step 3 Implement the **CYBSMposManagerDelegate** in your view controller.

Example CYBSMposManagerDelegate

```
class HomeViewController: UIViewController,
CYBSMposManagerDelegate {}
```

Step 4 Get the SDK shared instance.

Swift

```
let manager = CYBSMposManager.sharedInstance()
let settings = CYBSMposSettings(environment: .live, deviceID:
deviceID)
manager.settings = settings
manager.updateSettings()
```

Objective-C

```
CYBSMposManager *manager = [CYBSMposManager sharedInstance];

CYBSMposSettings *settings = [[CYBSMposSettings alloc]
initWithEnvironment:CYBSMposEnvironmentLive deviceID:"deviceId"];

manager.settings = settings;

[manager updateSettings];
```

Setting up and Connecting to the Terminal

You can set up the connection to the terminal using either the audio jack or Bluetooth.

Using the Audio Jack

Connect to the audio jack reader using **startAudio()** as shown below.

Swift

```
manager.startAudio(self)

//Callback methods

func onAudioDevicePlugged() {}

func onAudioInterrupted() {}
```

```
func onNoAudioDeviceDetected() {}
```

Objective-C

```
[manager startAudio:self];

//Callback methods

- (void)onAudioDevicePlugged { }

- (void)onAudioInterrupted { }

- (void)onNoAudioDeviceDetected { }
```

To disconnect from an audio jack reader, use **stopAudio()** as shown below.

Swift

```
manager.stopAudio(self)
```

Objective-C

```
[manager stopAudio:self];
```

Using Bluetooth

Use the following methods to set up for a Bluetooth connections. To troubleshoot problems, see "Bluetooth Terminal/Reader Connection Issues," page 71.

Scanning

Begin by scanning for connected Bluetooth devices. This method searches for connected Bluetooth terminal devices and returns a list of terminal devices connected to the mobile device in the **onBTReturnScanResults** callback method.

Swift

```
manager.scanBTDevices(self)

//Callback method

func onBTReturnScanResults(_ devices:[CYBSMposBluetoothDevice]? {

deviceList = devices

//loop devices array list

for device in deviceList {

let name = device.name

}

}
```

Objective-C

```
CYBSMposManager *manager = [[CYBSMposManager alloc]
initWithEnvironment: CYBSMposEnvironmentTest deviceID:@"your_
device_id"];

[manager scanBTDevices:self delegate:self];

//Callback method - CYBSMposManagerDelegate

- (void)deviceList:(nullable NSArray *) deviceList {

    if (nil == devices) {

        return;

    }

//    loop devices array list

    NSObject *deviceDict = [deviceList objectAtIndex:i];

    NSString *serial = [deviceDict valueForKey:@"serialNumber"];

    NSString *model = [deviceDict valueForKey:@"name"];

    NSString *version = [deviceDict
valueForKey:@"firmwareRevision"];
```

```
    }

}
```

Connecting

To connect a device from the returned-device list, use following code.

Swift

```
let selectedDevice = deviceList[0]
manager.connectBTDevice(selectedDevice)

//Callback method

func onBTConnected() {

manager.getDeviceInfo(self)

}
```

Objective-C

```
CYBSMposBluetoothDevice *device = self.deviceList[i];

[manager connectBTDevice:device];

//Callback method

- (void)onBTConnected {

[manager getDeviceInfo:self];

}
```

Disconnecting

To disconnect from a bluetooth device, use following code.

Swift

```
manager.disconenctBTDevice(self)

//Callback method

func onBTDisconnected(){ }
```

Objective-C

```
[manager disconenctBTDevice:self];

//Callback method

(void)onBTDisconnected {

}
```

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 decryption services type enum is defined in the *CYBSMposPaymentRequest.h* file as shown below.

```
typedef NS_ENUM(NSUInteger,
CYBSMposPaymentRequestDecryptionServices) {


CYBSMposPaymentRequestVISAHSM, // Visa HSM
CYBSMposPaymentRequestBluefin // Bluefin
```

```
};
```

The *CYBSMposPaymentRequest.h* keeps a property of decryption service enum value.

```
@property (nonatomic, assign)
CYBSMposPaymentRequestDecryptionServices decryptionService;
```

Calling the Visa HSM Service

Use the following code to call the Visa HSM service.

Swift

```
var amount:NSDecimalNumber = 10.0

let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.swipeOrInsertOrTap)

paymentRequest.decryptionService =
CYBSMposPaymentRequestDecryptionServices. VISAHSM
```

Objective C

```
NSDecimalNumber* amount = [[NSDecimalNumber alloc]
initWithString:@"10.0"];

CYBSMposPaymentRequest *request = [[CYBSMposPaymentRequest alloc]
initWithMerchantID:merchantID accessToken:accessToken
amount:amount
entryMode:CYBSMposPaymentRequestEntryModeSwipeOrInsertOrTap];

paymentRequest.decryptionService = CYBSMposPaymentRequestVISAHSM;
```

Calling the Bluefin Service

Swift

```
paymentRequest.decryptionService =
CYBSMposPaymentRequestDecryptionServices.bluefin;
```

Objective C

```
paymentRequest.decryptionService = CYBSMposPaymentRequestBluefin;
```

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 *CYBSMposPaymentRequest.h* file.

```
typedef NS_ENUM(NSUInteger, CYBSMposPaymentRequestEntryMode) {

    CYBSMposPaymentRequestEntryModeSwipe = 0, // Swipe

    CYBSMposPaymentRequestEntryModeSwipeOrInsertOrTap, // EMV -
contact, contactless.

   CYBSMposPaymentRequestEntryModeReaderKeyEntry, // Keyed-in card
data in the terminal.

    CYBSMposPaymentRequestEntryModeAppKeyEntry // Keyed-in card
data in the SDK.

};
```

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

Swift

```
let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.swipeOrInsertOrTap)
```

Objective C

```
CYBSMposPaymentRequest *request = [[CYBSMposPaymentRequest alloc]
initWithMerchantID:merchantID accessToken:accessToken
amount:amount
entryMode:CYBSMposPaymentRequestEntryModeSwipeOrInsertOrTap];
```

Magnetic Stripe Reader

Use **CYBSMposPaymentRequestEntryModeSwipe** for **entryMode** when initializing the **CYBSMposPaymentRequest** object.

Swift

```
let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.swipe)
```

Objective C

```
CYBSMposPaymentRequest *request = [[CYBSMposPaymentRequest alloc]
initWithMerchantID:merchantID accessToken:accessToken
amount:amount entryMode:CYBSMposPaymentRequestEntryModeSwipe];
```

Keyed-In in Transactions Using the Terminal Keypad

Use **CYBSMposPaymentRequestEntryModeReaderKeyEntry** when initializing **CYBSMposPaymentRequest** object.

Swift

```
let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.readerKeyEntry)
```

Objective C

```
CYBSMposPaymentRequest *request = [[CYBSMposPaymentRequest alloc]
initWithMerchantID:merchantID accessToken:accessToken
amount:amount
entryMode:CYBSMposPaymentRequestEntryModeReaderKeyEntry];
```

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 **CYBSMposPaymentRequestEntryModeAppKeyEntry** for **entryMode** when initializing the **CYBSMposPaymentRequest** object.

Swift

```
let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.appKeyEntry)
```

Objective C

```
CYBSMposPaymentRequest *request = [[CYBSMposPaymentRequest alloc]
initWithMerchantID:merchantID accessToken:accessToken
```

```
amount:amount
entryMode:CYBSMposPaymentRequestEntryModeAppKeyEntry];
```

Code Snippet

```
let manager = CYBSMposManager.sharedInstance()

let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID,

accessToken: accessToken, amount: amount, entryMode: .appKeyEntry)

             var cardData = VMposCardDataManual()

             cardData.accountNumber = "4111111111111111"

             let expMonth = "12"

             let expYear = "2018"

             cardData.expirationMonth = expMonth

             cardData.expirationYear = expYear

             cardData.cvNumber = "123"

             cardData.cardName = "test card"

             cardData.zipCode = "99508"

paymentRequest.manualEntryCardData = cardData

manager.performPayment(paymentRequest, parentViewController: self,
delegate: self)
```

Partial Authorization Transactions

Use the **partialIndicator** property of the **CYBSMposPaymentRequest** class if you want the transaction to be qualified for partial authorization. Following is the sample code snippet.

```
let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.swipeOrInsertOrTap)
paymentRequest.partialIndicator = true
```

If the transaction qualifies for partial authorization, SDK will prompt the merchant to accept or decline. Based on the selection, either a follow-on Capture or authorization reversal will be performed and the 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 68.

```
@interface CYBSMposPaymentResponse : NSObject

@property (nonatomic, strong, nullable) NSString *requestID;

@property (nonatomic, strong, nullable) NSString *subscriptionID;

@property (nonatomic, assign) CYBSMposPaymentResultDecision
decision;

@property (nonatomic, strong, nullable) NSString *reasonCode;

@property (nonatomic, strong, nullable) NSString *requestToken;

@property (nonatomic, strong, nullable) NSString
*authorizationCode;

@property (nonatomic, strong, nullable) NSDate
*authorizedDateTime;

@property (nonatomic, strong, nullable) NSString
*reconciliationID;

@property (nonatomic, strong, nullable) NSString
*merchantReferenceCode;

@property (nonatomic, strong, nullable)
CYBSMposPaymentResponseEmvReply *emvReply;
```

```
@property (nonatomic, strong, nullable)
CYBSMposPaymentResponseCard *card;

@property (nonatomic, strong, nullable) NSString
*authorizationMode;

@property (nonatomic, strong, nullable) NSString *entryMode;

@property (nonatomic, strong, nullable)
CYBSMposPaymentResponseAuthReply * authReply;
@property (nonatomic, strong, nullable)
CYBSMposPaymentResponseSubscriptionCreateReply *
paySubscriptionCreateReply;
@property (nonatomic, strong, nullable)
CYBSMposPaymentResponseCaptureReply * captureReply;
@property (nonatomic, strong, nullable)
CYBSMposPaymentResponseAuthReversalReply * authReversalReply;
@property (nonatomic, strong, nullable)
CYBSMposPaymentResponseVoidReply * voidReply;
@property (nonatomic, strong, nullable)
CYBSMposPaymentResponseRefundReply * refundReply;
@property (nonatomic, strong, nullable)
CYBSMposPaymentResponseICS_Message * ics_message;

@end

@interface CYBSMposPaymentResponseEmvReply : NSObject

@property (nonatomic, strong, nullable) NSMutableArray *emvTags;

+ (NSMutableArray *_Nonnull) buildEMVTags:(NSDictionary *_
Nonnull)emvDict;

- (NSString *_Nonnull)tagValue:(NSString *_
Nullable)iTagDescription;

@end

@interface CYBSMposPaymentResponseCard : NSObject

@property (nonatomic, strong, nullable) NSString *suffix;

@property(nonatomic, strong, nullable) NSString
*registeredApplicationProviderId;

@property (nonatomic, strong, nullable) NSString *maskedPAN;

@property (nonatomic, strong, nullable) NSString *expiryDate;

@property (nonatomic, strong, nullable) NSString *cardHolderName;
```

```
@end

@interface CYBSMposPaymentResponseAuthReply : NSObject

@property (nonatomic, strong, nullable) NSString * accountBalance;
@property (nonatomic, strong, nullable) NSString *
accountBalanceCurrency;
@property (nonatomic, strong, nullable) NSString *
accountBalanceSign;
@property (nonatomic, strong, nullable) NSString * amount;
@property (nonatomic, strong, nullable) NSString *
authorizationCode;
@property (nonatomic, strong, nullable) NSString *
authorizedDateTime;
@property (nonatomic, strong, nullable) NSString * avsCode;
@property (nonatomic, strong, nullable) NSString * avsCodeRaw;
@property (nonatomic, strong, nullable) NSString * ownerMerchantID;
@property (nonatomic, strong, nullable) NSString *
paymentNetworkTransactionID;
@property (nonatomic, strong, nullable) NSString * personalIDCode;
@property (nonatomic, strong, nullable) NSString * reasonCode;
@property (nonatomic, strong, nullable) NSString *
reconciliationID;
@property (nonatomic, strong, nullable) NSString * requestAmount;
@property (nonatomic, strong, nullable) NSString * requestCurrency;
@property (nonatomic, strong, nullable) NSString * transactionID;
@property (nonatomic, strong, nullable) NSString *
processorResponse;

@end

@interface CYBSMposPaymentResponseSubscriptionCreateReply :
NSObject

@property (nonatomic, strong, nullable) NSString * reasonCode;
@property (nonatomic, strong, nullable) NSString *subscriptionID;
@property (nonatomic, strong, nullable) NSString
*instrumentIdentifierID;
@property (nonatomic, strong, nullable) NSString
*instrumentIdentifierStatus;
@property (nonatomic, strong, nullable) NSString *
instrumentIdentifierNew;

@end

@interface CYBSMposPaymentResponseCaptureReply : NSObject

@property (nonatomic, strong, nullable) NSString * amount;
```

```
@property (nonatomic, strong, nullable) NSString *
processorTransactionID;
@property (nonatomic, strong, nullable) NSString * reasonCode;
@property (nonatomic, strong, nullable) NSString *
reconciliationID;
@property (nonatomic, strong, nullable) NSString * requestDateTime;

@end

@interface CYBSMposPaymentResponseAuthReversalReply : NSObject

@property (nonatomic, strong, nullable) NSString * amount;
@property (nonatomic, strong, nullable) NSString *
authorizationCode;
@property (nonatomic, strong, nullable) NSString * forwardCode;
@property (nonatomic, strong, nullable) NSString * reasonCode;
@property (nonatomic, strong, nullable) NSString *
reconciliationID;
@property (nonatomic, strong, nullable) NSString * requestDateTime;
@property (nonatomic, strong, nullable) NSString *
processorResponse;

@end

@interface CYBSMposPaymentResponseVoidReply : NSObject

@property (nonatomic, strong, nullable) NSString * amount;
@property (nonatomic, strong, nullable) NSString * currency;
@property (nonatomic, strong, nullable) NSString * reasonCode;
@property (nonatomic, strong, nullable) NSString * requestDateTime;

@end

@interface CYBSMposPaymentResponseRefundReply : NSObject

@property (nonatomic, strong, nullable) NSString * amount;
@property (nonatomic, strong, nullable) NSString * forwardCode;
@property (nonatomic, strong, nullable) NSString * reasonCode;
@property (nonatomic, strong, nullable) NSString *
reconciliationID;
@property (nonatomic, strong, nullable) NSString * requestDateTime;

@end

@interface CYBSMposPaymentResponseICS_Message : NSObject

@property (nonatomic, strong, nullable) NSString * auth_rmsg;
@property (nonatomic, strong, nullable) NSString * auth_auth_
amount;
```

```
@property (nonatomic, strong, nullable) NSString * auth_payment_
network_transaction_id;
@property (nonatomic, strong, nullable) NSString * auth_auth_avs;
@property (nonatomic, strong, nullable) NSString * bill_trans_ref_
no;
@property (nonatomic, strong, nullable) NSString * bill_bill_trans_
ref_no;
@property (nonatomic, strong, nullable) NSString * ics_rflag;
@property (nonatomic, strong, nullable) NSString * terminal_id;
@property (nonatomic, strong, nullable) NSString * auth_rcode;
@property (nonatomic, strong, nullable) NSString * bill_rflag;
@property (nonatomic, strong, nullable) NSString * ics_decision_
reason_code;
@property (nonatomic, strong, nullable) NSString * bill_bill_
request_time;
@property (nonatomic, strong, nullable) NSString * ics_rmsg;
@property (nonatomic, strong, nullable) NSString * bill_bill_
amount;
@property (nonatomic, strong, nullable) NSString * merchant_ref_
number;
@property (nonatomic, strong, nullable) NSString * bill_rmsg;
@property (nonatomic, strong, nullable) NSString * bill_return_
code;
@property (nonatomic, strong, nullable) NSString * request_token;
@property (nonatomic, strong, nullable) NSString * acquirer_
merchant_number;
@property (nonatomic, strong, nullable) NSString * auth_auth_time;
@property (nonatomic, strong, nullable) NSString * ics_return_code;
@property (nonatomic, strong, nullable) NSString * auth_payment_
service_data_2;
@property (nonatomic, strong, nullable) NSString * bill_rcode;
@property (nonatomic, strong, nullable) NSString * auth_rflag;
@property (nonatomic, strong, nullable) NSString * request_id;
@property (nonatomic, strong, nullable) NSString * auth_trans_ref_
no;
@property (nonatomic, strong, nullable) NSString * currency;
@property (nonatomic, strong, nullable) NSString * auth_auth_
response;
@property (nonatomic, strong, nullable) NSString * auth_return_
code;
@property (nonatomic, strong, nullable) NSString * auth_auth_code;

@end
```

Implementing Your Use Cases

Settings with Secure Acceptance—the services **enum** is defined in the *CYBSPaymentRequest.h* file.

```
typedef NS_ENUM(NSUInteger,
CYBSMposPaymentRequestSupportedServices) {

CYBSMposPaymentRequestTokenized = 1,

CYBSMposPaymentRequestEndlessAisle = 2,

CYBSMposPaymentRequestTokenizedEndlessAisle = 3,

CYBSMposPaymentRequestRetail = 6,

CYBSMposPaymentRequestTokenizedRetail = 7

};
```

Additional Attributes Supported in the SDK

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

```
let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.swipeOrInsertOrTap)

paymentRequest.merchantReferenceCode = “Cybersource”
```

The **CYBSMposPaymentRequest** class has a **currency** parameter, which takes currency in the form of currency code. For example, USD, INR, or EUR. The default currency code is USD.

```
let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.swipeOrInsertOrTap)
```

```
paymentRequest.purchaseTotal.currency = “INR”
```

CYBSMposPaymentRequest class has an ECI indicator enumeration parameter **commerceIndicator** of type **CYBSMposPaymentRequestCommerceIndicator**.

```
let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.swipeOrInsertOrTap)

paymentRequest.commerceIndicator = .retail
```

CYBSMposPaymentRequest class has an enumeration parameter **paymentService** of type **CYBSMposPaymentRequestSupportedServices**.

```
let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.swipeOrInsertOrTap)

paymentRequest.paymentService = .tokenized
```

CYBSMposPaymentRequest class has an additional **merchantTransactionIdentifier,** parameter which is a unique for every transaction.

```
let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.swipeOrInsertOrTap)

paymentRequest.merchantTransactionIdentifier = “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 **CYBSMposPaymentRequestTokenized**, the card data is tokenized.

Swift

```
paymentRequest.paymentService = .tokenized
```

Objective C

```
paymentRequest.paymentService =
CYBSMposPaymentRequestTokenized
```

Collecting Billing and Shipping Information

Billing and shipping address can be set using **CYBSMposAddress** class as shown below:

Swift

```
let billTo = CYBSMposAddress()

billTo.street1 = "billing address"

billTo.city = "city"

billTo.state = "state"

billTo.postalCode = "zip"

billTo.email = "email"
```

```
let shipTo = CYBSMposAddress()

shipTo.street1 = "shipping address"

shipTo.city = "city"

shipTo.state = "state"

shipTo.postalCode = "zip"

shipTo.email = "email"

let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.swipeOrInsertOrTap)

paymentRequest.shippingAddress = shipTo

paymentRequest.billingAddress = billTo
```

Objective C

```
CYBSMposAddress *billTo = [[CYBSMposAddress alloc] init];

billTo.street1 = @"billing address";

billTo.city = @"city";

billTo.state = @"state";

billTo.postalCode = @"zip";

billTo.email = @"email";

CYBSMposAddress *shipTo = [[CYBSMposAddress alloc] init];

shipTo.street1 = @"shipping address";

shipTo.city = @"city";

shipTo.state = @"state";

shipTo.postalCode = @"zip";

shipTo.email = @"email";

CYBSMposPaymentRequest *request = [[CYBSMposPaymentRequest alloc]
initWithMerchantID:merchantID accessToken:accessToken
```

```
amount:amount
entryMode:CYBSMposPaymentRequestEntryModeSwipeOrInsertOrTap];

request.billingAddress = billTo;

request.shippingAddress = shipTo;
```

Adding Merchant-Defined Data

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

```
@property (copy, nonatomic) NSArray * _Nullable
merchantDefinedDataArray;
```

Swift

```
let mdd = ["data0", "data1", "data2", "data3", "data4"]

paymentRequest. merchantDefinedDataArray = mdd
```

Objective C

```
NSArray* mdd = @[@"data0", @"data1", @"data2", @"data3", @"data4"];

    paymentRequest.merchantDefinedDataArray = mdd;
```

Printing the Transaction Receipt

If the device has built-in printer, the SDK will print a receipt that is compliant with EMVCo and card brand requirements. Implement the following delegate methods to show printing receipt status:

```
// returns the result of print

- (void)onReturnReceiptPrintResult:(BOOL)result;

// on completion of print data

- (void)onReceiptPrintDataEnd;

// on print data cancelled

- (void)onReceiptPrintDataCancelled;
```

The above code is applicable for Wisepad2 plus printer. The BBPOS Wisepad2 Plus printer has the capability to print physical receipts. We have a default print structure for you designed in the SDK. For other devices, construct your receipt with SDK response.

Generating Your Own Receipt

After the transaction, a payment response object is return in the callback method.

```
- (void)performPaymentDidFinish:(nullable CYBSMposPaymentResponse *)result

            error:(nullable NSError *)error;
```

With the **CYBSMposPaymentResponse** object, you can generate your own receipt page.

Generating Your Email Receipt

You can generate your own email receipt using the following code.

Swift

```
let manager = CYBSMposManager.sharedInstance()

let receiptRequest = CYBSMposReceiptRequest(transactionID:
"transactionID",

  toEmail: "emailId",

  accessToken: "accessToken")

manager.sendReceipt(receiptRequest, delegate: self)
```

Objective-C

```
CYBSMposManager *manager = [CYBSMposManager sharedInstance];

CYBSMposReceiptRequest *request = [[CYBSMposReceiptRequest alloc]

  initWithTransactionID:@"transactionId"

  toEmail:@"emailID"

  accessToken:@"accessToken"];

[manager sendReceipt:request delegate:self];
```

The **sendReceipt** result is returned with the following delegate method:

```
- (void)sendReceiptDidFinish:(nullable NSDictionary *)result

                       error:(nullable NSError *)error;
```

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

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

```
var itemsList = Array<CYBSMposItem>()

   let lineItem1 = CYBSMposItem()

   lineItem1.name = “iPhone 6S”

   lineItem1.quantity = 1

   lineItem1.price = 499.9

   itemsList.append(lineItem1)


   let lineItem2 = CYBSMposItem()

   lineItem2.name = “iPhone X”

   lineItem2.quantity = 1

   lineItem2.price = 599.0

   itemsList.append(lineItem2)

let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.swipeOrInsertOrTap)

paymentRequest.items =  itemsList
```

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

```
let merchantData = CYBSMerchantDescriptor()

   merchantData.merchantDescriptorBusinessName = "CyberSource mPOS
Store"

   merchantData.merchantDescriptorStreet = "901 Metro center Blvd"

   merchantData.merchantDescriptorCity = "Foster City"
```

```
merchantData.merchantDescriptorState = "CA"

merchantData.merchantDescriptorCountry = "USA"

merchantData.merchantDescriptorPostalCode = "94404"

merchantData.merchantDescriptorPhoneNumber = "650-302-7012"

merchantData.merchantDescriptorEmail = "usriniva@visa.com"


let manager = CYBSMposManager.sharedInstance()

    manager.updateMerchantDescriptorSettings(merchantData)
```

This **CYBSMerchantDescriptor** class can also be used for transaction request as shown below:

```
let paymentRequest = CYBSMposPaymentRequest(merchantID:
merchantID, accessToken: accessToken, amount: amount, entryMode:
.swipeOrInsertOrTap)

paymentRequest.merchantData = merchantData
```

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

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

```
let dateFormatter = DateFormatter()

var query = CYBSMposTransactionSearchQuery

 query.dateFrom = dateFormatter.date(from:
dateFormatter.string(from:

 Date()))!.timeIntervalSince1970

          query.dateTo = query.dateFrom + 86400


 let sharedInstance = CYBSMposManager.sharedInstance()

 manager.performTransactionSearch(self.query, accessToken:
accessToken,

 delegate: self)
```

Transaction search will be returned with delegate method:

```
func performTransactionSearchDidFinish(_ result:
CYBSMposTransactionSearchResult?, error: Error?)
```

Voiding a Transaction

```
let sharedInstance = CYBSMposManager.sharedInstance()

let voidRequest = CYBSMposVoidRequest(merchantID: merchantID,
accessToken: accessToken, merchantReferenceCode: referenceCode,
transactionID: transactionID)
```

```
manager.performVoid(voidRequest, delegate: self)
```

Void transaction result will be returned with delegate method:

```
- (void)performVoidDidFinish:(nullable CYBSMposPaymentResponse
*)result

  error:(nullable NSError *)error;
```

Refunding a Transaction

```
let sharedInstance = CYBSMposManager.sharedInstance()

let refundRequest = CYBSMposRefundRequest(merchantID: merchantID,
accessToken: accessToken, merchantReferenceCode: referenceCode,
transactionID: transactionID, currency: “USD”, amount: 10.0)

manager.performRefund(voidRequest, delegate: self)
```

The Refund transaction result will be returned with the following delegate method:

```
- (void)performRefundDidFinish:(nullable CYBSMposPaymentResponse
*)result

  error:(nullable NSError *)error;
```

Partially Refunding a Transaction

To process a partial refund transaction, navigate to the History screen of the application and search for transactions. Select a transaction to view the transaction details and select Partial Refund. Enter the amount you want to refund and click **OK**. The amount is refunded and the transaction details are updated. For a code snippet, see "Refunding a Transaction," page 53.

Capturing an Authorization for the Endless Aisle Use Case

```
let sharedInstance = CYBSMposManager.sharedInstance()

 let captureRequest = CYBSMposCaptureRequest(merchantID:
merchantID, accessToken: accessToken!,

merchantReferenceCode: referenceCode, transactionID:
transactionID,

currency: currency, amount: amount)

  manager.performCapture(captureRequest, delegate: self)
```

Capturing Authorization result will be returned with delegate method:

```
- (void)performCaptureDidFinish:(nullable CYBSMposPaymentResponse
*)result

  error:(nullable NSError *)error;
```

Authorization Reversal

```
let sharedInstance = CYBSMposManager.sharedInstance()

let authReversalRequest =
CYBSMposAuthorizationReversalRequest(merchantID: merchantID,
accessToken: accessToken,

merchantReferenceCode: referenceCode, transactionID:
transactionID,
currency: currency, amount: amount)

manager.performAuthorizationReversal(authReversalRequest,
delegate: self)
```

Authorization reversal result will be returned with delegate method:

```
- (void)performAuthorizationReversalDidFinish:(nullable
CYBSMposPaymentResponse *)result

  error:(nullable NSError *)error;
```

Retrieving a Signature

A signature can be retrieved using the **getTransactionSignature()** API, which uses the following parameters:

- **transactionID**—The transaction ID of the signature being retrieved.
- **accessToken**—OAuth access token.
- **delegate**—Callback interface of type **CYBSMposManagerDelegate.**

```
- (void)getTransactionSignature:(nonnull NSString *)transactionID

accessToken:(nonnull NSString *)accessToken

delegate:(nullable id<CYBSMposManagerDelegate>)delegate
```

Example

```
let sharedInstance = CYBSMposManager.sharedInstance();

sharedInstance.getTransactionSignature(“transactionId”,

accessToken: “accessToken” delegate: self)
```

The signature data will be sent back to the application using the delegate method **getTransactionSignatureDidFinish()**.

- **signature**: Signature data.

- **error**: API error, if any.

```
- (void)getTransactionSignatureDidFinish:(nullable NSData
*)signature

error:(nullable NSError *)error;
```

Skipping the Signature

There is a **skipSignature** property defined in the **CYBSMposPaymentRequest** class:

```
@property (nonatomic, assign) BOOL skipSignature;
```

After initializing the **CYBSMposPaymentRequest** object, set **skipSignature** to `true` to skip the signature panel shown at the end of the transaction processing flow, which comes from the SDK. Set `false` to show signature. The default value is `false`.

You can apply your own logic to hide or show signature. For example, if the amount is greater than or equal to 10, set false, otherwise true.

```
let minAmountForSignature = 10

var skipSignatureView = false

if(amount.compare(NSDecimalNumber(value: minAmountForSignature))
== .orderedAscending){

   skipSignatureView = true

}
```

Overriding the Terminal ID and Alternate Terminal ID

The CYBSMposManager object keeps a reference of the **CYBSMposSettings** instance.

```
@property (nonatomic, copy, nonnull) CYBSMposSettings *settings;
```

To set the terminal ID, alternate terminal ID and merchant ID, you must set the following properties to the **CYBSMposSettings** instance, then update the **settings** instance to **CYBSMposManager**. Start a transaction with this **CYBSMposManager** instance, your merchant ID (MID), **terminalIDAlternate** and **terminalID** will be returned in the payment request.

```
@property (nonatomic, copy, nullable) NSString *mid;

@property (nonatomic, copy, nullable) NSString *terminalID;

@property (nonatomic, copy, nullable) NSString
*terminalIDAlternate;
```

Designing a User Interface

To customize your own transaction processing page, you can initialize the **CYBSMposUISettings** object and set its values.

```
let uiSettings = CYBSMposUISettings()

// background color change

uiSettings.backgroundColor = UIColor.green;

// logo

if let imageData = try? Data(contentsOf: imageURL) {

   uiSettings.topImage = UIImage(data: imageData)

}

// transaction label font

 uiSettings.regularFontName = "GillSans-Italic";

 let manager = CYBSMposManager.sharedInstance()
```

```
manager.uiSettings = uiSettings
```

Registering the Device from the Sample Application

To register a device from the sample application:

Step 1 Open sample application and navigate to **Tab > Devices**.

Step 2 Enter credentials such as **merchant id**, **client id**, **user name**, and **password**.

Step 3 Enter the ID of the device you want to register. If you do not enter an ID, the application will create one for this device.

Step 4 Click **Register Device**. The device is registered and its status is set to **Pending**.

Step 5 To activate the registered device, log in to CyberSource Business Center, navigate to **Device Management > Mobile Devices**, and click **Activate**.

Note

If you have already entered these values in the Settings screen, the application pre-fills these values.

Code Snippets

Device registration using username and password:

```
let manager = CYBSMposManager.sharedInstance()
manager.activateDevice(withClientId: “clientId”, withUserName:
“username” withPassword: “password”, withMerchantId: “merchantId”,
withDeviceId: “deviceId”, withDescription: nil,
withDevicePlatform: nil, withSDKVersion: nil, withAppVersion: nil,
withPhoneNumber: nil, with:self)
```

Device Registration using client credentials:

```
let manager = CYBSMposManager.sharedInstance()
manager.activateDevice(withClientId:“clientId”, withMerchantId:
“merchantId”, withDeviceId: “deviceId”, withClientSecret:
“clientSecret”, withDescription: nil, withDevicePlatform: nil,
withSDKVersion: nil, withAppVersion: nil, withPhoneNumber: nil,
with: self)
```

The device registration result will be returned with following delegate method:

```
func onDeviceRegister(_ responseData: Error) { }
```

Setting Up Multiple User Accounts

You can set up multiple user accounts at one time using a JSON file. This comes in handy if you have multiple stores and you want to administer them separately.

Create a .json file, for example, *StoresConfiguration.json,* with the following key value pairs, keeping the keys the same, but using your own values.

```
{
    "Store1_Test" :
    [
        {
            "Store_Name" : "Store1",
            "CyberSource_MID": "store1_test_mid",
            "Device_ID": "store1_test_device_id",
            "Client_ID": "fG41gMU2qc",
            "Terminal_ID": ""
        }
    ],
    "Store1_Production" :
    [
        {
            "Store_Name" : "Store1",
            "CyberSource_MID": "store1__live_mid",
            "Device_ID": "store1_live_device_id",
            "Client_ID": "FAq1E4NaTI",
            "Terminal_ID": ""
        }
    ],
"Store2_Test" :
    [
```

```
        {
            "Store_Name" : "Store2",
            "CyberSource_MID": "store2_test_mid",
            "Device_ID": "store2_test_device_id",
            "Client_ID": "rW4ZfMU24c",
            "Terminal_ID": ""
        }
    ],
    "Store2_Production" :
    [
        {
            "Store_Name" : "Store2",
            "CyberSource_MID": "store2_live_mid",
            "Device_ID": "store2_live_device_id",
            "Client_ID": "ESC4E7NrEI",
            "Terminal_ID": ""
        }
    ],

}
```

Configuring the Stores

To configure the stores:

Step 1 Open the sample application.

Step 2 Connect the iPhone or iPad to the iTunes application and navigate to the **File Sharing** tab from the connected device.

Step 3 Select the CyberSource mPOS application from the list of **Apps** and save *StoresConfiguration.json* to your local machine.

Step 4 Update the store configuration information.

Step 5 Select the **Add** option and replace that file again.

Step 6 In the mPOS application, navigate to **Settings** and choose the store you want to use.

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.

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 In-Person SDK provides **CYBSMposManager** class with following methods.

```
(void)checkForOTAUpdateIsTestReader:(BOOL)isTestReader
   withOTAUpdateStatus:(OTACheckUpdateRequiredBlock _
   Nonnull)iUpdateCheckBlock;
```

- **isTestReader**: `true` for test readers, `false` otherwise.
- **iUpdateCheckBlock**: this block will be executed when the SDK finds the status of the update. The application will be notified if the firmware or configuration is updated.

```
- (void)startOTAUpdateIsTestReader:(BOOL)isTestReader
   withProgress:(OTAUpdateProgressBlock _Nonnull)iOTAProgressBlock
   andOTACompletionBlock:(OTACheckUpdateCompletedBlock _
   Nonnull)iUpdateCompleteBlock;
```

- **isTestReader**: `true` for test readers, `false` otherwise.
- **iOTAProgressBlock:** This block will be executed when there is a change in the progress of the update. The application is notified about the progress of the update.
- **iUpdateCheckBlock**: This block will be executed when the SDK finds the status of the update. The application is notified if the firmware or configuration is updated.

```
- (void) startConfigurationUpdateIsTestReader:(BOOL)isTestReader

   withProgress:(OTAUpdateProgressBlock _Nonnull)iOTAProgressBlock

   andOTACompletionBlock:(OTACheckUpdateCompletedBlock _

   Nonnull)iUpdateCompleteBlock;
```

- **isTestReader**: `true` for test readers, `false` otherwise.
- **iOTAProgressBlock:** This block will be executed when there is a change in the progress of the update. The application is notified about the progress of the update.
- **iUpdateCheckBlock**: This block will be executed when the SDK finds the status of the update. The application will be notified when the configuration is updated.

```
- (void)startFirmwareUpdateIsTestReader:(BOOL)isTestReader

readerType:(OTAReaderType)iReaderType

withProgress:(OTAUpdateProgressBlock _Nonnull)iOTAProgressBlock

andOTACompletionBlock:(OTACheckUpdateCompletedBlock _
Nonnull)iUpdateCompleteBlock;
```

- **isTestReader**: `true` for test readers, `false` otherwise.
- **iOTAProgressBlock:** This block will be executed when there is a change in the progress of the update. The application is notified about the progress of the update.
- **iUpdateCheckBlock**: This block will be executed when the SDK finds the status of the update. The application will be notified when the configuration is updated.

OTA Update Sample Code

Use the following steps to perform an OTA update using the In-Person SDK.

Note

The following sample code assumes reader connectivity mode as Bluetooth and reader type as demo (test) reader.

Step 1 Connect to a reader using Bluetooth or the audio jack.

Step 2 Check if any update is available by calling the **checkForOTAUpdateIsTestReader ()** method.

```
let sharedInstance = CYBSMposManager.sharedInstance();

sharedInstance?.check(forOTAUpdateIsTestReader:

!self.isTestReader) { (iFirmwareUpdateRequired,
iConfigurationUpdateRequired, iErrorCode:CYBSMposDeviceErrorCode,
iErrorString, iStatusCode:CYBSMposOTAResultCode, iStatusMessage)in

if (iErrorCode == CYBSMposDeviceErrorCode.noError) {

   if (iStatusCode == CYBSMposOTAResultCode.success) {

      if (iFirmwareUpdateRequired || iConfigurationUpdateRequired)
{

         print("update required")

         print("Check for update", message: "Update required")

      } else {

         print ("Check for update", message: "Your reader is upto
date")

      }

   } else {

      print ("Check for update error", message: iStatusMessage!)

   }

} else {

   print ("Check for update error", message: iErrorString!)
```

```
}
```

Step 3 If an update is required, call the **startOTAUpdateIsTestReader ()** method to update both firmware and configuration. The following code snippet will update both:

```
let sharedInstance = CYBSMposManager.sharedInstance();

sharedInstance?.startOTAUpdateIsTestReader(!self.isTestReader,
withProgress: { (iPercentage:Float, iUpdateType:OTAUpdateType) in

   // Progress in percentage

   }, andOTACompletionBlock: { (iUpdateSuccessful:Bool,
iErrorCode:CYBSMposDeviceErrorCode, iErrorString,
iStatusCode:CYBSMposOTAResultCode, iStatusMessage) in

   // Update completed

   if (iUpdateSuccessful && iErrorCode ==
CYBSMposDeviceErrorCode.noError)

   {

      print("Reader Updated successfully!", message: "Note: Please
make sure reader is restarted properly before using it for next
transaction.")

}  else {

   if let errorMsg = iErrorString {

      print("Update failed", message: errorMsg)

   else {

   print("Update Status", message: iStatusMessage!)

   }

   }

})
```

Step 4 If only a firmware update is required, call the **startFirmwareUpdateIsTestReader ()** method to update the firmware.

```
let sharedInstance = CYBSMposManager.sharedInstance();

sharedInstance?.startFirmwareUpdateIsTestReader
(!self.isTestReader, withProgress: { (iPercentage:Float,
iUpdateType:OTAUpdateType) in

   // Progress in percentage

   }, andOTACompletionBlock: { (iUpdateSuccessful:Bool,
iErrorCode:CYBSMposDeviceErrorCode, iErrorString,
iStatusCode:CYBSMposOTAResultCode, iStatusMessage) in

   // Update completed

   if (iUpdateSuccessful && iErrorCode ==
CYBSMposDeviceErrorCode.noError)

   {

      print("Reader Updated successfully!", message: "Note: Please
make sure reader is restarted properly before using it for next
transaction.")

   } else {

   if let errorMsg = iErrorString {

      print("Update failed", message: errorMsg)

   } else {

      print("Update Status", message: iStatusMessage!)

   }

   }

})
```

Step 5 If only a configuration update is required, call the **startConfigurationUpdateIsTestReader ()** method to update the configuration.

```
let sharedInstance = CYBSMposManager.sharedInstance();

sharedInstance?. startConfigurationUpdateIsTestReader
(!self.isTestReader, withProgress: { (iPercentage:Float,
iUpdateType:OTAUpdateType) in

   // Progress in percentage

   }, andOTACompletionBlock: { (iUpdateSuccessful:Bool,
iErrorCode:CYBSMposDeviceErrorCode, iErrorString,
iStatusCode:CYBSMposOTAResultCode, iStatusMessage) in

   // Update completed

   if (iUpdateSuccessful && iErrorCode ==
CYBSMposDeviceErrorCode.noError)

   {

      print("Reader Updated successfully!", message: "Note: Please
make sure reader is restarted properly before using it for next
transaction.")

   } else {

      if let errorMsg = iErrorString {

         print("Update failed", message: errorMsg)

   } else {

      print("Update Status", message: iStatusMessage!)

   }

   }

})
```

Setting Reader Connection Type

The default connection type in the SDK is Bluetooth. If you use an audio jack reader, you can use the following method to change the connection type:

```
let sharedInstance = CYBSMposManager.sharedInstance();
```

```
sharedInstance?.setConnectionMode(VMposEMVConnectionMode.audio)
```

To set it back to Bluetooth mode, use:

```
sharedInstance?.setConnectionMode(VMposEMVConnectionMode.bluetooth
)
```

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

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 iOS/ macOS 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 the that 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

SDK Pre-Installation Checks

iOS version - 9.0 and later.

Cocopods - 1.2.1

Bluetooth Terminal/Reader Connection Issues

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

- Connect the reader to our demo app and see if the connection is successful. Bluetooth readers must be paired in the mobile Bluetooth settings. In the demo app, click “scan for the readers”. Select the terminal /reader you intend to use.
- Add Bluetooth usage description to *info.plist.*
- Add supported external accessory protocols to *info.plist*.
- Sample value for the Wisepad 2 reader should be **com.bbpos.bt.wisepad**.
- Add the necessary framework to your project.
- Need to add **NSBluetoothPeripheralUsageDescription** in *Info.plist* as described in the link below:

https://developer.apple.com/documentation/corebluetooth

UsageDescription provides the terminal user information about Bluetooth connection for accessing the terminals.

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.