Mobile Software Development Kit
Integrator's Manual

© 2017 Direct Connect ™

Preliminary

Table of Contents

Preliminary

Preliminary

Figures

Tables

Preliminary

Revisions

Date	Author	Comments
04/01/17	Francois Bergeon	Initial draft
04/05/17	Francois Bergeon	Added Data Adapter classes
04/07/17	Francois Bergeon	Split device managers into their own libraries
04/15/17	Francois Bergeon	Added class descriptions
04/26/17	Francois Bergeon	Added iOS asynchronous operations. Clarified synchronous and asynchronous preferences.
05/02/17	Francois Bergeon	Added timeout to Request.Process synchronous methods
05/26/17	Francois Bergeon	Added IDTech BTMag and UniPay devices. Added PAX devices. Added Context parameter to the DeviceManager contructor (Android only).
05/31/17	Francois Bergeon	Added IDTech UniMag devices.
06/08/17	Francois Bergeon	Added note for UniMag devices on Android. Refactored Device class properties.
06/21/17	Francois Bergeon	Added Swift examples and discussion. Added Swift method definitions.
07/12/17	Francois Bergeon	Added documentation for the provided sample app.

Preliminary

1. Introduction

Scope

The purpose of this technical document is to document usage and best practces of the Direct Connect Software Development Kit (SDK) that enables mobile application developers to interface to the Direct Connect Gateway.

Conventions & nomenclature

The reader is expected to be familiar with the Unified Markup Language (UML) terminology. The UML syntax used loosely follows the UML 2.0 standards. The following item nomenclature has been adopted throughout the document:

- Actor names are defined as "*the<ActorName>*"they are represented in *italics* in the body of the document.
- Business use cases are defined as "B<number>_<name>".
- Architecture use cases are defined as "A<number>_<name>".
- Design use cases are defined as D<number>_<component>_<name>.

Preliminary

2. Design Philosophy

The Direct Connect Mobile SDK is designed specifically for ease of integration. Both Android and iOS versions share the same architecture and similar object model designs.

Choice is given to the developer to interface with the SDK in either a synchronous or asynchronous way whenever possible. Due to operating system philosophy, asynchronous operations are preferred on the Android platform.

For asynchronous operations, the observer design pattern is implemented by the way of listener interfaces on Android and delegate protocols or completion blocks on iOS.

Preliminary

3. Object Model

The Direct Connect Mobile SDK is composed of transaction processing classes that expose services offered by the Direct Connect Gateway, and device support classes that allow a mobile application to interact with a user-facing interaction device.

Transaction Processing

The Direct Connect Mobile SDK exposes a simple object model composed of a request class and a response class. For example, the `CreditCardRequest` class and a `CreditCardResponse` class implement the gateway's ProcessCreditCard methods.

A `CreditCardRequest.Listener` class is defined to handle "on-processed" events for asynchronous operations (Figure 1).

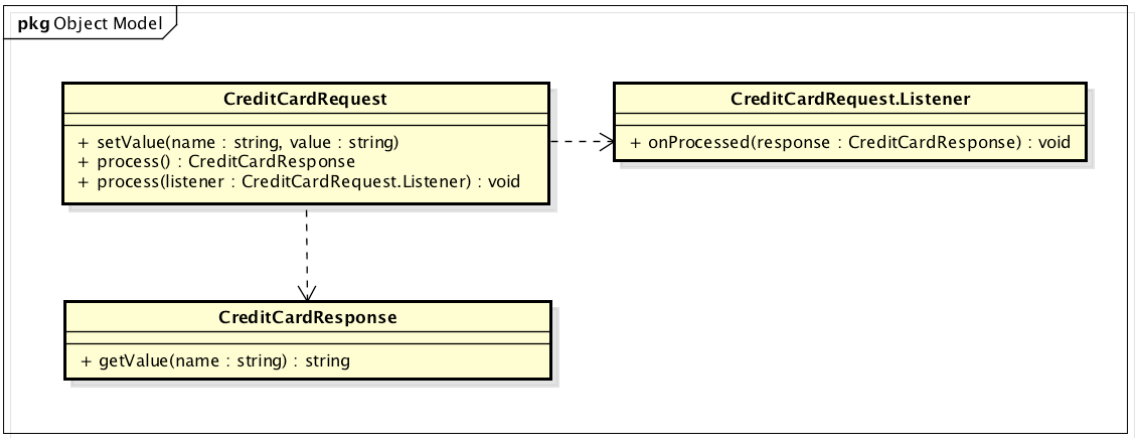

Figure 1: ProcessCreditCard class diagram

Preliminary

Android/Java transaction code sample

The following Java code snippet is a sample for synchronously processing a credit card transaction with the Direct Connect Gateway (Figure 2). This example is for illustration purposes only: synchronous operations should not be performed on the application's main thread.

```
CreditCardRequest req = new CreditCardRequest(endPoint);
req.setValue(CreditCardRequest.UserName, "username");
req.setValue(CreditCardRequest.Password, "password");
req.setValue(CreditCardRequest.TransType, "Sale");
req.setValue(CreditCardRequest.CardNum, "4012111111111111");
req.setValue(CreditCardRequest.ExpDate, "0420");
req.setValue(CreditCardRequest.Amount, "11.00");

CreditCardResponse resp = req.process();

String result = resp.getValue(CreditCardResponse.Result);
String respMsg = resp.getValue(CreditCardResponse.RespMSG);
String authCode = resp.getValue(CreditCardResponse.AuthCode);
...
```

Figure 2: Java synchronous processing sample

For asynchronous operations, a `CreditCardRequest.Listener` is passed as a parameter to the `process()` method to receive "on-processed" events as shown in the example below (Figure 3):

```
CreditCardRequest req = new CreditCardRequest(endPoint);
req.setValue(CreditCardRequest.UserName, "username");
req.setValue(CreditCardRequest.Password, "password");
req.setValue(CreditCardRequest.CardNum, "4012111111111111");
req.setValue(CreditCardRequest.ExpDate, "0420");
req.setValue(CreditCardRequest.Amount, "11.00");

req.process(new CreditCardRequest.Listener() {
   public void onProcessed(CreditCardResponse response) {
      String result = response.getValue(CreditCardResponse.Result);
      String respMSG = response.getValue(CreditCardResponse.RespMSG);
      String authCode = response.getValue(CreditCardResponse.AuthCode);
      ...
   }
});
```

Figure 3: Java asynchronous processing sample

Preliminary

iOS/Swift transaction code sample

The following Swift code snippet is a sample for synchronously processing a credit card transaction on iOS (Figure 4):

```
let req = DCGCreditCardRequest(endPoint)!
req.setValue("username", forKey:DCGCreditCardRequest.UserName)
req.setValue("password", forKey:DCGCreditCardRequest.Password)
req.setValue("Sale", forKey:DCGCreditCardRequest.TransType)
req.setValue("4012111111111111", forKey:DCGCreditCardRequest.CardNum)
req.setValue("0420", forKey:DCGCreditCardRequest.ExpDate)
req.setValue("11.00", forKey:DCGCreditCardRequest.Amount);

let resp = req.process()!

let result = (resp.getValue(DCGCreditCardResponse.Result) as! Int)
let respMSG = (resp.getValue(DCGCreditCardResponse.RespMSG) as! String)
let authCode = (resp.getValue(DCGCreditCardResponse.AuthCode) as! String)
...
```

Figure 4: Swift synchronous processing sample

For asynchronous operations, a `DCGCreditCardDelegate` is passed as the "`delegate`" parameter of the `process()` method to receive "on-processed" events. Alternatively a completion block that receives the response object can be passed as the "`completion`" parameter of the `process()` method as shown in the example below (Figure 5):

```
let req = DCGCreditCardRequest(endPoint)!
req.setValue("username", forKey:DCGCreditCardRequest.UserName)
req.setValue("password", forKey:DCGCreditCardRequest.Password)
req.setValue("Sale", forKey:DCGCreditCardRequest.TransType)
req.setValue("4012111111111111", forKey:DCGCreditCardRequest.CardNum)
req.setValue("0420", forKey:DCGCreditCardRequest.ExpDate)
req.setValue("11.00", forKey:DCGCreditCardRequest.Amount)

let req = process(completion: {
     (resp: DCGCreditCardResponse!) in
     let result = (resp.getValue(DCGCreditCardResponse.Result) as! Int)
     let respMSG = (resp.getValue(DCGCreditCardResponse.RespMSG) as! String)
     let authCode = (resp.getValue(DCGCreditCardResponse.AuthCode) as! String)
     ..
} as (DCGCreditCardResponse?) -> Void)
```

Figure 5: Swift asynchronous processing sample

Preliminary

iOS/Objective-C transaction code sample

The following Objective-C code snippet is a sample for synchronously processing a credit card transaction on iOS (Figure 6):

```
DCGCreditCardRequest *req = [[DCGCreditCardRequest alloc] init:endPoint];
[req setValue:@"username" forKey:DCGCreditCardRequest.UserName];
[req setValue:@"password" forKey:DCGCreditCardRequest.Password];
[req setValue:@"Sale" forKey:DCGCreditCardRequest.TransType];
[req setValue:@"4012111111111111" forKey:DCGCreditCardRequest.CardNum];
[req setValue:@"0420" forKey:DCGCreditCardRequest.ExpDate];
[req setValue:@"11.00" forKey:DCGCreditCardRequest.Amount];

DCGCreditCardResponse *resp = [req process];

NSString *result = [[resp getValue:DCGCreditCardResponse.Result] stringValue];
NSString *respMSG = [resp getValue:DCGCreditCardResponse.RespMSG];
NSString *authCode = [resp getValue:DCGCreditCardResponse.AuthCode];
...
```

Figure 6: Objective-C synchronous processing sample

For asynchronous operations, a `DCGCreditCardDelegate` is passed as a parameter to the `processWithDelegate` method to receive "on-processed" events. Alternatively a completion block that receives the response object can be passed as a parameter to the `processWithCompletion` method as shown in the example below (Figure 7):

```
DCGCreditCardRequest *req = [[DCGCreditCardRequest alloc] init:endPoint];
[req setValue:@"username" forKey:DCGCreditCardRequest.UserName];
[req setValue:@"password" forKey:DCGCreditCardRequest.Password];
[req setValue:@"Sale" forKey:DCGCreditCardRequest.TransType];
[req setValue:@"4012111111111111" forKey:DCGCreditCardRequest.CardNum];
[req setValue:@"0420" forKey:DCGCreditCardRequest.ExpDate];
[req setValue:@"11.00" forKey:DCGCreditCardRequest.Amount];

[req processWithCompletion:^void (DCGCreditCardResponse *resp) {
     NSString *result = [[resp getValue:DCGCreditCardResponse.Result] stringValue];
     NSString *respMSG = [resp getValue:DCGCreditCardResponse.RespMSG];
     NSString *authCode = [resp getValue:DCGCreditCardResponse.AuthCode];
     ...
}];
```

Figure 7: Objective-C asynchronous processing sample

Preliminary

Device Support

In addition to the transaction processing classes detailed above, the Direct Connect Mobile SDK also supports a variety of input devices by exposing the `DeviceManager` base class, the `DeviceManager.Listener` delegate interface and the `DeviceManager.Device`, `DeviceManager.CardData` and `DeviceManager.PINData` helper classes (Figure 8).

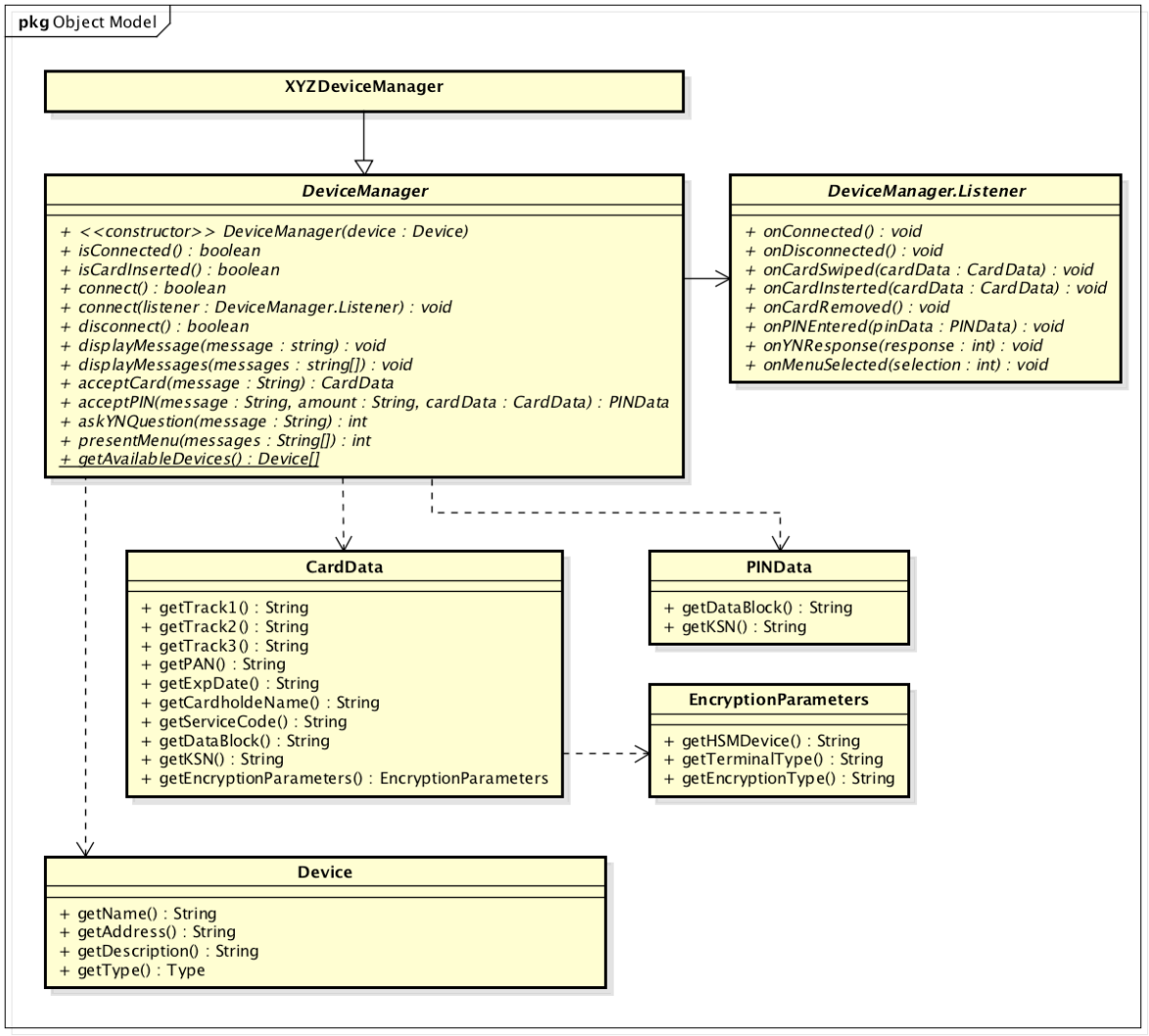

Figure 8: DeviceManager class diagram

Android/Java device code sample

The following Java code snippet is a sample for synchronously obtaining a credit card swipe from an XYZ-brand device on Android (Figure 9).

```
Device[] devices = DeviceManagerXYZ.getAvailableDevices();
DeviceManagerXYZ deviceManager = new DeviceManagerXYZ(devices[0], view.getContext());
if (deviceManager.connect()) {
   deviceManager.acceptCard("Swipe Card");
   CardData cardData = deviceManager.acceptCard();
   String pan = cardData.getPAN();
   String expDate = cardData.getExpDate());
   ...
}
```

Figure 9: Java DeviceManager synchronous sample

For asynchronous operations, a `DeviceListener` object is passed to the `DeviceManager` `connect` method. The following Java code snippet is a sample for asynchronously obtaining a credit card swipe from a device (Figure 10).

```
deviceManager.connect(new DeviceManager.Listener() {
   public void onConnected() {
      deviceManager.acceptCard("Swipe Card");
   }

   public void onCardSwiped(CardData cardData) {
      String pan = cardData.getPAN();
      String expDate = cardData.getExpDate());
      ...
   }

   public void onPINEntered(PINData pinData) {}
   ...
// Other required event handlers
   ...
   public void onDisconnected() {}
};
```

Figure 10: Java DeviceManager asynchronous sample

Preliminary

iOS/Swift device code sample

The following Swift code snippet is a sample for obtaining a credit card swipe from an XYZ-brand device on iOS (Figure 11):

```
let devices = (DCGXYZDeviceManager.getAvailableDevices()! as NSArray)
let device = (devices.object(0) as! DCGDevice)
let deviceManager = DCGXYZDeviceManager(device)!
if (deviceManager.connect())! {
     let cardData = (deviceManager.acceptCard("Swipe Card"))!
     let pan = cardData.PAN;
     let expDate = cardData.ExpDate;
     ...
}
```

Figure 11: Swift DeviceManager synchronous sample

For asynchronous operations, a delegate object is passed to the `connect()` method. The following Swift code snippet is a sample for asynchronously obtaining a credit card swipe from a device (Figure 12):

```
class MyClass : XYZBaseClass, DCGDeviceDelegate

func myMethod() {
     ...
     let devices = (DCGXYZDeviceManager.getAvailableDevices()! as NSArray)
     let device = (devices.object(0) as! DCGDevice)
     let deviceManager = DCGXYZDeviceManager(device)!
     deviceManager.connect(self)
     ...
}

func onConnected() {
    deviceManager.acceptCard("Swipe Card...")
}

func onCardSwiped(_ cardData:DCGCardData!) {
     let pan = cardData.PAN
     let expDate = cardData.ExpDate
     ...
}

// Other required event handlers
...
```

Figure 12: Swift DeviceManager asynchronous sample

Preliminary

iOS/Objective-C device code sample

The following Objective-C code snippet is a sample for synchronously obtaining a credit card swipe from an XYZ-brand device on iOS (Figure 13):

```
NSArray *devices = [DCGXYZDeviceManager getAvailableDevices];
DCGXYZDevice *device = [devices objectAtIndex:0];
DCGXYZDeviceManager *deviceManager = [[DCGXYZDeviceManager alloc] init:device];
if ([deviceManager connect]) {
     DCGCardData *cardData = [deviceManager acceptCard:@"Swipe Card"];
     NSString *pan = cardData.PAN;
     NSString *expDate = cardData.ExpDate;
     ...
}
```

Figure 13: Objective-C DeviceManager synchronous sample

For asynchronous operations, a delegate object is passed to the connect method. The following Objective-C code snippet is a sample for asynchronously obtaining a credit card swipe from a device (Figure 14):

```
@interface MyClass : <DCGDeviceDelegate>
     ...
@end


@implementaion MyClass

-(void)myMethod {
     ...
     [deviceManager connect:self]
     ...
}

-(void)onConnected {
    [deviceManager acceptCard:@"Swipe Card..."];
}

-(void)onCardSwiped:(DCGCardData *)cardData {
     NSString *pan = cardData.PAN;
     NSString *expDate = cardData.ExpDate;
     ...
}

// Other required event handlers
   ...
@end
```

Figure 14: Objective-C DeviceManager asynchronous sample

Preliminary

Data Adapters

A series of data adapter classes are offered to simplify integration of the Request/Response classes with the Device Support classes.

With this approach, instead of manually copying card-related data from the `CardData` object to the `Request` object by implementing logic based on how the card data was obtained (magnetic track, P2PE, etc), an adapter class can automatically populate the `Request` object with the appropriate data elements from a `CardData` object (Figure 15). The subsequent sample code snippets illustrate how a `CreditCardAdapter` can be used along with a `CardData` object to populate a `CreditCardRequest` object (Figure 16, Figure 17, Figure 18).

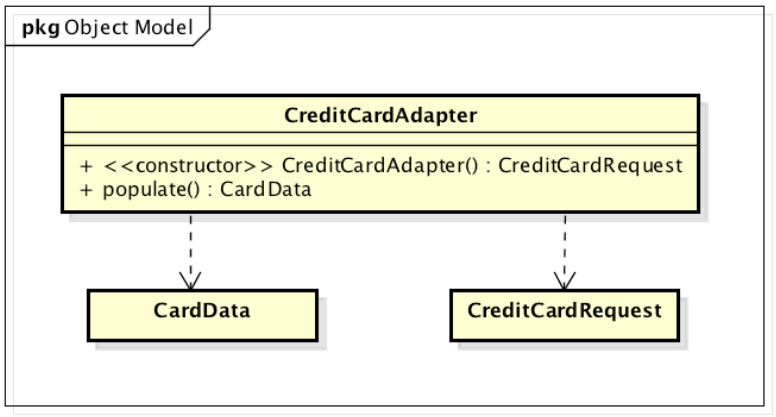

Figure 15: Adapter class diagram

```
CreditCardRequest request = new CreditCardReques(endPoint);
...
deviceManager.connect(new DeviceManager.Listener() {
   ...
   public void onCardSwiped(CardData cardData) {
      CreditCardAdapter adapter = new CreditCardAdapter(request);
      adapter.populate(cardData);
      ...
   }
   ...
};
```

Figure 16: Java CreditCardAdapter sample

Preliminary

```
let req = DCGCreditCardRequest(endpoint);
...
if (deviceManager.connect()) {
    let cardData = deviceManager.acceptCard("Swipe Card")!
    let adapter = DCGCreditCardAdapter(req)!
    adapter.populate(cardData)
    ...
}
```

Figure 17: Swift CreditCardAdapter sample

```
DCGCreditCardRequest *req = [[DCGCreditCardRequest alloc] init:endPoint];
...
if ([deviceManager connect]) {
    DCGCardData *cardData = [deviceManager acceptCard:@"Swipe Card"];
    DCGCreditCardAdapter *adapter = [[DCGCreditCardAdapter alloc] init:req];
    [adapter populate:cardData];
    ...
}
```

Figure 18: Objective-C CreditCardAdapter sample

Preliminary

4. Transaction Processing

The following classes have been implemented to expose the transaction processing services offered by the Direct Connect Gateway:

Request class	Response class	Data adapter class
CreditCardRequest	CreditCardResponse	CreditCardAdapter
DebitCardRequest*	DebitCardResponse*	DebitCardAdapter*
EBTCardRequest*	EBTCardResponse*	EBTCardAdapter*
GiftCardRequest*	GiftCardResponse*	GiftCardAdapter*
CheckRequest*	CheckResponse*	CheckAdapter*

*reserved for future use

Table 1: Transaction request, response and data adapter classes

Please refer to the following tables (Table 2, Table 3, Table 4) and the DC Gateway API Developer's Guide for a list of input and output parameters supported by the various request and response classes.

Please note that sub-classed parameters have been flattened into their top-level class. For example, the `AltMerchantName` member of the `ExtData` sub-class used by the `ProcessCreditCard` method is exposed as the "`AltMerchantName`" parameter, directly supported by the `CreditCardRequest`'s `setValue()` method. There is no need to reference the `ExtData` sub-class in the parameter name.

CreditCardRequest data elements								
TransType / Element	Auth / Sale	RepeatSale	Force	Return	Reversal	Void	Capture	CaptureAll
UserName	R	R	R	R	R	R	R	R
Password	R	R	R	R	R	R	R	R
TransType	R	R	R	R	R	R	R	R
CardNum	R[1]	R[1]	O[1]	R[1]	R[1]	O[1]	O[1]	O[1]
ExpDate	R[1]	R[1]	O[1]	R[1]	R[1]	O[1]	O[1]	O[1]
MagData	R[1]	R[1]	O[1]	R[1]	R[1]	O[1]	O[1]	O[1]
NameOnCard	O	O	O	O	O	O	O	O
Amount	R	R	R	R	R	O	O	O
InvNum	O	O	O	O	O	O	O	O
PNRef	O	O	R	R	R	R	R	O
Zip	O	O	O	O	O	O	O	O
Street	O	O	O	O	O	O	O	O
CVNum	O	O	O	O	O	O	O	O
1: Either CardNum/ExpDate or MagData								

Table 2: CreditCardRequest data elements

Preliminary

CreditCardRequest extended data elements	
AltMerchName	O
AltMerchAddr	O
AltMerchCity	O
AltMerchState	O
AltMerchZip	O
AuthCode	O
Authentication	O
BillPayment	O
BillToState	O
BypassAvsCvv	O
CardPresent	O
City	O
Clinical_Amou	O
ConvenienceAm	O
CustCode	O
CustomerID	O
CVPresence	O
Dental_Amount	O
EmvData	O
EntryMode	O[2]
ExternalIP	O
Force	O
IIAS_Indicator	O
Level3Amt	N/A*
PartialIndicator	O
PONum	O
QHP_Amount	O
RegisterNum	O
RX_Amount	O
SequenceNum	O
SequenceCount	O
ServerID	O
Target	O
TaxAmt	O
Timeout	O
TipAmt	O
TrainingMode	O
TransactionID	O
Vision_Amount	O
*: Not supported in the current version 2: Required if MagData or EMVData fields are populated	

Table 3: CreditCardRequest extended data elements

Preliminary

CreditCardRequest P2PE data elements
HSMDevice
TerminalType
EncryptionType
DataBlock
KSN

Table 4: CreditCardRequest P2PE data elements

Preliminary

5. Device Support

Device support is currently implemented as a series of classes derived from a `DeviceManager` base class (Table 5).

DeviceManager derived classes
VirtualDeviceManager
MiuraDeviceManager
BTMagDeviceManager
UniMagDeviceManager
UniPayDeviceManager*
PAXDeviceManager*

*reserved for future use

Table 5: Device manager derived classes

The `VirtualDeviceManager` class exposes a software-only input device simulator that interacts with the user for testing. Other classes expose an interface to their corresponding hardware devices. Due to its interactions with the application's UI thread, synchronous operations are not supported with the `VirtualDeviceManager`.

The `MiuraDeviceManager` class exposes an interface to the Miura line of Bluetooth devices.

The `BTMagDeviceManager`, `UniMagDeviceManager` and `UniPayDeviceManager` classes expose interfaces to various members of the IDTech line of devices.

The `PAXDeviceManager` class exposes an interface to the PAX line of devices.

Preliminary

6. Android Integration

On the Android/Java platform, transaction-related classes are defined in the `com.directconnect.mobilesdk.transaction` package; device-related classes are defined in the `com.directconnect.mobilesdk.device` package; adapter classes are defined in the `com.directconnect.mobilesdk.adapter` package.

Network connectivity must be enabled in the project's manifest by adding the following tag to the `<manifest>` element:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Please refer to the `DCMobileSDK_Sample` Android Studio project for a more complete example of Android integration.

NOTE: Due to system architecture, ***asynchronous*** operations are the preferred mode of operation on the Android platform. Synchronous network or device calls performed on the application's main thread can lead to an exception or an unresponsive state.

Libraries

Android Studio projects need to include the following libraries that should be copied in the module's `libs` directory (`[`*project*`]/[`*module*`]/libs`):

`DCMobileSDK.aar`

The module's `build.gradle` file also needs to be changed as follows:

Add the following block:

```
repositories {
   flatDir {
      dirs 'libs'
   }
}
```

Add the following lines to the `dependencies` block:

```
compile (name: 'DCMobileSDK', ext:'aar')
```

In addition to the above-detailed steps, device-specific libraries need to be included in the project if external devices are to be used. Note that the input device simulator exposed by the `VirtualDeviceManager` class is included in the basic library referenced above and does not require any additional libraries.

Preliminary

For interfacing with Miura devices:

- Add the following libraries to the module's `libs` directory:
  ```
  DCMobileSDK-Miura.aar
  Lib-Miura-SDK.aar
  ```
- Add the following lines to the `dependencies` block of the module's `build.gradle` file:
  ```
  compile (name: 'DCMobileSDK-Miura', ext:'aar')
  compile (name: 'Lib-Miura-SDK', ext:'aar')
  ```

For interfacing with IDTech devices

- Add the following libraries to the module's `libs` directory (replace `<version>` with the actual SDK library version):
  ```
  DCMobileSDK-IDT.aar
  ```
- Add the following lines to the `dependencies` block of the module's `build.gradle` file:
  ```
  compile (name: 'DCMobileSDK-IDT', ext:'aar')
  ```

NOTE: Due to the variety of Android host devices, the first connection to a phone-jack UniMag reader may take a substantial amount of time to let the driver auto-configure for the device's specific audio settings. In addition, the "Microphone" and "Storage" permissions must be enabled for the installed Android app, either by acknowledging the request at install time or by manually authorizing in Settings -> Application -> <application> -> Permissions.

For interfacing with PAX devices

Add the following libraries to the module's `libs` directory (replace `<version>` with the actual Easylink SDK library version):

```
DCMobileSDK-PAX.aar
EasyLinkSdk_<version>.jar
```

Add the following lines to the `dependencies` block of the module's `build.gradle` file:

```
compile (name: 'DCMobileSDK-PAX', ext:'aar')
```

Preliminary

7. iOS Integration

In iOS class names are prefixed with "DCG" (Direct Connect Gateway) in accordance with the platform's best practices for naming conventions.

If the application interfaces with a Bluetooth device, remember to authorize communication in the Xcode project's settings by enabling the following capabilities:

<Project name> -> Capabilities -> Background Modes
- External accessory communication
- Uses Bluetooth LE accessories*

(*if Bluetooth Low-Energy devices are to be used)

Please refer to the `DCMobileSDK_SwiftSample` and the `DCMobileSDK_ObjCSample` Xcode projects for a more complete example of iOS integration.

Libraries

iOS Xcode projects need to include the following libraries:

`libDCMobileSDK.a`

The `DCMobileSDK/include` directory needs to be added to the project's Header Search Paths.

For Swift projects, a bridging header file is required to access the Direct Connect Mobile SDK libraries written in Objective-C. The bridging header can be created locally, or the provided `DCMobileSDK-Bridging-Header.h` header file can be referenced as the project's "Objective-C Bridging Header" under the "Swift Compiler – General" project settings.

For Objective-C projects, transaction-specific header files (e.g. `DCMobileSDK-CreditCard.h`) can be included to access all transaction-related and device-related classes.

In addition to the above-detailed steps, device-specific libraries need to be included in the project if external devices are to be used. Note that the input device simulator exposed by the `VirtualDeviceManager` class is included in the basic library referenced above and does not require any additional libraries.

Preliminary

For interfacing with Miura devices:

- Add the following libraries to the Xcode project:
  - libDCMobileSDK-Miura.a
  - libMiuraSDK.a
- Add the following lines to the <dict> element in the project's Info.plist file:

```
<key>UISupportedExternalAccessoryProtocols</key>
<array>
  <string>com.miura.shuttle</string>
  <string>com.miura.rpi</string>
</array>
```

For interfacing with IDTech devices:

For all IDTech devices:

- Add the following libraries to the Xcode project:
  - libDCMobileSDK-IDT.a

For BTMag (Bluetooth) devices:

- Add the following libraries to the Xcode project:
  - libBTMagSDK.a
- Add the following lines to the <dict> element in the project's Info.plist file:

```
<key>UISupportedExternalAccessoryProtocols</key>
<array>
  <string> com.idtechproducts.BTMag</string>
</array>
```

For UniMag (phone jack) devices:

- Add the following libraries to the Xcode project:
  - libIDTECH_UniMag.a
- Add the following frameworks to the Xcode project:
  - MediaPLayer.framework
  - AudioToolbox.framework
  - AVFoundation.framework
- Add the following lines to the <dict> element in the project's Info.plist file:

```
<key>NSMicrophoneUsageDescription</key>
<string>Uses audio jack for card swipe</string>
```

For interfacing with PAX devices:

Add the following libraries:

- libDCMobileSDK-PAX.a
- libPaxEasylinkController.a

Preliminary

8. Class Reference

Request classes

Description

Implementation classes derived from the base Request class, designed to model a request for a specific payment type.

Android mplementations

- CreditCardRequest
- DebitCardRequest*
- EBTCardRequest*
- GiftCardRequest*
- CheckRequest*

* Reserved for future use

iOS mplementations

- DCGCreditCardRequest
- DCGDebitCardRequest*
- DCGEBTCardRequest*
- DCGGiftCardRequest*
- DCGCheckRequest*

* Reserved for future use

Methods

setValue	
Prototypes	
Android	
`void setValue(String name, String value)`	
`void setValue(String name, int value)`	
`void setValue(String name, boolean value)`	
iOS/Swift	
`Void setValue(value: Any!, forKey: String!)`	
iOS/Objective-C	
`(void)setValue:(id)value forKey:(NSString *)name`	
Description	
Set the value of a request element.	
Parameters	
`name`	Element name – see section 4 (Transaction Processing) for a list of valid element names
`value`	Value to set
Returns	
N/A	

Preliminary

process	
Prototypes	
Android	
Synchronous	`XYZResponse process()`
	`XYZResponse process(int timeout)`
Asynchronous	`void process(XYZRequest.Listener listener)`
`XYZResquest` and `XYZResponse` are implementations of the `Resquest` and `Response` abstract classes	
iOS/Swift	
Synchronous	`DCGXYZResponse! process()`
	`DCGXYZResponse! Process(timeout: UInt)`
Asynchronous	`Void process(delegate: DCGXYZDelegate!)`
	`Void process(completion: ((DCGXYZResponse?) -> Void)!)`
iOS/Objective-C	
Synchronous	`(DCGXYZResponse *)process`
	`(DCGXYZResponse *)processWithTimeout:(NSUInteger)timeout`
Asynchronous	`(void)processWithDelegate:(id<DCGXYZDelegate>)delegate`
	`(void)processWithCompletion:(void (^)(DCGXYZResponse *))completion`
On `iOS`, `DCGXYZResponse` is an implementation of the `DCGResponse` abstract class. `DCGXYZDelegate` is the delegate protocol associated with the `DCGXYZRequest`.	
Description	
Process the payment request. If no timeout is specified for a synchronous call, an infinite timeout is used.	
Parameters	
`timeout`	(optional, synchronous) Timeout in seconds for synchronous operations
`listener`	(Android, asynchronous) Listener instance for the Request class
`delegate`	(iOS, asynchronous) Delegate instance for the Request class
`completion`	(iOS, asynchronous) Completion block for the Request class
Returns	
(Synchronous) Populated instance of the corresponding response class	

Preliminary

Response classes

Description

Implementation class derived from the base Response class, designed to model a response to a request for a specific payment type.

Android implementations

- CreditCardResponse
- DebitCardResponse*
- EBTCardResponse*
- GiftCardResponse*
- CheckResponse*

• Reserved for future use

iOS implementations

- DCGCreditCardResponse
- DCGDebitCardResponse*
- DCGEBTCardResponse*
- DCGGiftCardResponse*
- DCGCheckResponse*

* Reserved for future use

Methods

getValue	
Prototypes	
Android	
`String getValue(String name)`	
`int getInt(String name)`	
`boolean getBool(String name)`	
iOS/Swift	
`Any! getValue(name: String!)`	
iOS/Objective-C	
`(id)getValue:(NSString *)name`	
Description	
Get the value of a response element.	
Parameters	
`name`	Element name – see section 4 (Transaction Processing) for a list of valid element names
Returns	
The value of the element	

Preliminary

Listener and delegate classes

Description

Abstract delegate class designed to receive an on-processed event upon completion of a payment request. Class to be implemented by the developer.

Android interfaces

- `CreditCardRequest.Listener`
- `DebitCardRequest.Listener*`
- `EBTCardRequest.Listener*`
- `GiftCardRequest.Listener*`
- `CheckRequest.Listener*`

* Reserved for future use

iOS protocols

- `DCGCreditCardDelegate`
- `DCGDebitCardDelegate*`
- `DCGEBTCardDelegate*`
- `DCGGiftCardDelegate*`
- `DCGCheckDelegate*`

* Reserved for future use

Methods

onProcessed	
Prototypes	
Android	
`void onProcessed(XYZResponse response)`	
Where `XYZResponse` is an implementation of the abstract Response class	
iOS/Swift	
`Void onProcessed(_ response:DCGXYZResponse!)`	
iOS/Objective-C	
`(void)onProcessed:(DCGXYZResponse *)response`	
Where `DCGXYZResponse` is an implementation of the abstract DCGResponse class	
Description	
Called when asynchronous request processing is complete	
Parameters	
`response`	Populated instance of the Response class
Returns	
N/A	

Preliminary

Device class

Description

Utility class holding device identification properties.

Android implementation

- `Device`

iOS implementation

- `DCGDevice`

Properties

Prototypes
Android
`String get<property>()`
iOS/Swift
`String! <property>`
iOS/Objective-C
`(NSString *)<property>`
Where <property> is the requested property (see below)
Description
Get the value of the associated property
Parameters
None
Returns
The value of the associated property

Property	Property name	
Name	`Name`	Device name
Address	`Address`	Device address or identifier
String description	`toString (Android)` `description (iOS)`	String representation of the device (name & address)
Device type	`Type`	Class type of the associated DeviceManager

Preliminary

DeviceManager classes

Description

Implementation classes derived from the base `DeviceManager` class, designed to interface with a specific device type.

Android implementations

- `VirtualDeviceManager`
- `MiuraDeviceManager` (in `DCMobileSDK-miura` library)
- `BTMagDeviceManager` (in `DCMobileSDK-IDT` library)
- `UniMagDeviceManager` (in `DCMobileSDK-IDT` library)
- `UniPayDeviceManager` (in `DCMobileSDK-IDT` library)
- `PAXDeviceManager` (in `DCMobileSDK-PAX` library)

iOS implementations

- `DCGVirtualDeviceManager`
- `DCGMiuraDeviceManager` (in `DCMobileSDK-miura` library)
- `DCGBTMagDeviceManager` (in `DCMobileSDK-IDT` library)
- `DCGUniMagDeviceManager` (in `DCMobileSDK-IDT` library)
- `DCGUniPayDeviceManager` (in `DCMobileSDK-IDT` library)
- `DCPAXDeviceManager` (in `DCMobileSDK-PAX` library)

Methods

Constructor	
Prototypes	
Android	
`DeviceManager(Device device, Context context)`	
iOS/Swift	
`DeviceManager(device:DCGDevice!)`	
iOS/Objective-C	
`(id)init:(DCGDevice *)device`	
Description	
Instantiate a new device manager object to interface with a device of the supported type	
Parameters	
`device`	Device object
`context`	Application context (Android only)

Preliminary

get AvailableDevices	
Prototypes	
Android (static)	
`Device[] getAvailableDevices()`	
iOS/Swift	
`[Any!] getAvailableDevices()`	
iOS/Objective-C	
`(NSArray *)getAvailableDevices`	
Description	
Get an array populated with all available devices of the supported type.	
Parameters	
None	
Returns	
Array of device objects	

connect	
Prototypes	
Android	
Synchronous	`boolean connect()`
Asynchronous	`void connect(DeviceManager.Listener listener)`
iOS/Swift	
Synchronous	`Bool connect()`
Asynchronous	`Void connect(delegate: DCGDeviceDelegate!)`
iOS/Objective-C	
Synchronous	`(Boolean)connect`
Asynchronous	`(void)connect:(DCGDeviceDelegate *)delegate`
Description	
Connect to the device. Choice of the synchronous or asynchronous call to this method dictates the behavior of subsequent calls to the following methods. In other words, if a Listener or Delegate object is specified, subsequent calls will be asynchronous, if no Listener or Delegate object is specified, subsequent calls will be synchronous.	
Parameters	
`listener`	(Android, asynchronous) Listener instance
`delegate`	(iOS, asynchronous) Delegate instance
Returns	
(Synchronous) True if connected, False otherwise	

Preliminary

disconnect
Prototypes
Android
`Boolean disconnect()`
iOS/Swift
`Bool disconnect()`
iOS/Objective-C
`(Boolean)disconnect`
Description
Connect to the device
Parameters
None
Returns
True if connected (synchronous), False otherwise or asynchronous

Preliminary

displayMessage	
Prototypes	
Android	
`void displayMessage(String message)`	
iOS/Swift	
`Void displayMessage(message: String!)`	
iOS/Objective-C	
`(void)displayMessage:(NSString *)message`	
Description	
Display message on the device	
Parameters	
`message`	Message to be displayed
Returns	
N/A	

displayMessages	
Prototypes	
Android	
`void displayMessages(String[] messages)`	
iOS/Swift	
`Void displayMessages(messages: [Any]!)`	
iOS/Objective-C	
`(void)displayMessages:(NSMutableArray *)messages`	
Description	
Display messages on the device	
Parameters	
`messages`	Messages to be displayed
Returns	
N/A	

Preliminary

acceptCard	
Prototypes	
Android	
`CardData acceptCard(String message)`	
iOS/Swift	
`DCGCardData! acceptCard(message: String!)`	
iOS/Objective-C	
`(DCGCardData *)acceptCard:(NSString *)message`	
Description	
Display message and accept card	
Parameters	
`message`	Message to be displayed
Returns	
Synchronous	Populated CardData object. A null value indicates the operation timed-out or was cancelled by the user. A non-null CardData object with `DataType.nil` is indicative of a bad swipe.
Asynchronous	Null

Preliminary

acceptPIN	
Prototypes	
Android	
`PINData acceptPIN(String message, String amount, CardData cardData)`	
iOS/Swift	
`DCGPINData! acceptPIN(message: String!, amount: String!, cardData: DCGCardData!)`	
iOS/Objective-C	
`(DCGPINData *)acceptPIN:(NSString *)message withAmount:(NSString *)amount withCardData:(DCGCardData *)cardData`	
Description	
Display message and accept PIN entry	
Parameters	
`message`	Message to be displayed
`amount`	Transaction amount
`cardDara`	CardData object associated with the card
Returns	
Synchronous	Populated PINData object. A null value indicates the operation timed-out or was cancelled by the user. If the PINData object is not null but its DataBlock property is null, PIN entry was bypassed by the user.
Asynchronous	Null

Preliminary

askYNQuestion	
Prototypes	
Android	
`int askYNQuestion(String message)`	
iOS/Swift	
`Int32 askYNQuestion(message String!)`	
iOS/Objective-C	
`(int)askYNQuestion:(NSString *)message`	
Description	
Display message and accept Yes/No entry	
Parameters	
`message`	Message to be displayed
Returns	
1 for yes, 2 for no, -1 for cancel (synchronous) or -1 (asynchronous)	

presentMenu	
Prototypes	
Android	
`int presentMenu(String[] messages)`	
iOS/Swift	
`Int32 presentMenu(messages: [Any]!)`	
iOS/Objective-C	
`(int)presentMenu:(NSMutableArray *)messages`	
Description	
Display menu and accept selection	
Parameters	
`messages`	Messages to be displayed
Returns	
Selected menu index (synchronous) or -1 (asynchronous)	

Preliminary

isConnected
Prototypes
Android
`boolean isConnected()`
iOS/Swift
`Bool isConnected()`
iOS/Objective-C
`(Boolean)isConnected`
Description
Return connection status
Parameters
None
Returns
True if connected, false otherwise

isCardInserted
Prototypes
Android
`boolean isCardInserted()`
iOS/Swift
`Bool isCardInserted()`
iOS/Objective-C
`(Boolean)isCardInserted`
Description
Return card presence status
Parameters
None
Returns
True if card inserted, false otherwise

Preliminary

DeviceManagerListener/DCGDeviceDelegate classes

Description

Abstract delegate class designed to receive asynchronous device events. Class to be implemented by the developer.

Android interfaces

- `DeviceManager.Listener`

iOS protocols

- `DCGDeviceDelegate`

Methods

onConnected
Prototypes
Android
`void onConnected()`
iOS/Swift
`Void onConnected()`
iOS/Objective-C
`(void)onConnected`
Description
Receive connected event
Parameters
None
Returns
N/A

onDisconnected
Prototypes
Android
`void onDisonnected()`
iOS/Swift
`Void onDisonnected ()`
iOS/Objective-C
`(void)onDisonnected`
Description
Receive disconnected event
Parameters
None
Returns
N/A

onCardSwiped	
Prototypes	
Android	
`void onCardSwiped(CardData cardData)`	
iOS/Swift	
`Void onCardSwiped(_ cardData: DCGCardData!)`	
iOS/Objective-C	
`(void)onCardSwiped:(DCGCardData *)cardData`	
Description	
Receive card swiped event	
Parameters	
`cardData`	Populated CardData object. A null value indicates the operation timed-out or was cancelled by the user. A non-null CardData object with `DataType.nil` is indicative of a bad swipe.
Returns	
N/A	

onPINEntered	
Prototypes	
Android	
`void onPINEntered(PINData pinData)`	
iOS/Swift	
`Void onPINEntered (_ pinData: DCGPINData!)`	
iOS/Objective-C	
`(void)onPINEntered:(DCGPINData *)pinData`	
Description	
Receive PIN entered swiped event	
Parameters	
`pinData`	Populated PINData object. A null value indicates the operation timed-out or was cancelled by the user. If the PINData object is not null but its DataBlock property is null, PIN entry was bypassed by the user.
Returns	
N/A	

Preliminary

onYNAnswered	
Prototypes	
Android	
`void onYNAnswered(int response)`	
iOS/Swift	
`Void onYNAnswered(_ response: Int32)`	
iOS/Objective-C	
`(void)onYNAnswered:(int)response`	
Description	
Receive Y/N response event	
Parameters	
`response`	1 for yes, 0 for no, -1 for cancel
Returns	
N/A	

onMenuSelected	
Prototypes	
Android	
`void onMenuSelected(int selection)`	
iOS/Swift	
`Void onMenuSelected (_ response: Int32)`	
iOS/Objective-C	
`(void)onMenuSelected:(int)selection`	
Description	
Receive menu selected event	
Parameters	
`selection`	Menu index
Returns	
N/A	

Preliminary

CardData class

Description

Utility class holding card data.

Android implementation

- `CardData`

iOS implementation

- `DCGCardData`

Properties

Prototypes
Android
`String get<property>()`
iOS/Swift
`String! <property>`
iOS/Objective-C
`(NSString *)<property>`
Where <property> is the requested property (see below)
Description
Get the value of the associated property
Parameters
None
Returns
The value of the associated property

Property	Property name	
Track 1	`Track1`	Card's track 1 data
Track 2	`Track2`	Card's track 2 data
Track 3	`Track3`	Card's track 3 data
Primary Account Number	`PAN`	Card's primary account number
Expiration Date	`ExpDate`	Card's expiration date "MMYY"
Service Code	`ServiceCode`	Card's 3-digit service code
Cardholder Name	`CardholderName`	Cardholder name
Data Block	`DataBlock`	Encrypted card data
Key Serial Number	`KSN`	DUKPT key serial number
Encryption Parameters	`EncryptionParameters`	P2PE encryption parameters
Data Type	`DataType`	Card data type

Preliminary

EncryptionParameters class

Description

Utility class holding P2PE encryption parameters.

Android implementation

- `EncryptionParameters`

iOS implementation

- `DCGEncryptionParameters`

Properties

Prototypes
Android
`String get<property>()`
iOS/Swift
`String! <property>`
iOS/Objective-C
`(NSString *)<property>`
Where <property> is the requested property (see below)
Description
Get the value of the associated property
Parameters
None
Returns
The value of the associated property

Property	Property name	
HSM Device Identifier	`HSMDevice`	Identifier of the target HSM device
Terminal Type	`TerminalType`	Terminal type
Encryption Type	`EncryptionType`	Encryption type

Preliminary

PINData class

Description

Utility class holding PIN parameters.

Android implementation

- `PINData`

iOS implementation

- `DCGPINData`

Properties

Prototypes
Android
`String get<property>()`
iOS/Swift
`String! <property>`
iOS/Objective-C
`(NSString *)<property>`
Where <property> is the requested property (see below)
Description
Get the value of the associated property
Parameters
None
Returns
The value of the associated property

Property	Property name	
Data block	`DataBlock`	DUKPT-encrypted PIN
Key Serial Number	`KSN`	DUKPT key serial number

Preliminary

9. Direct Connect Mobile SKD sample app

The Direct Connect Mobile SDK is packaged with a sample app for Android and iOS platforms. The purpose of the sample app is to illustrate a typical integration with the SDK as described in this document, and should in no way be considered as a form of guidance or best practice when developing a production app.

By default, the provided sample app interfaces with the Virtual Device Manager. If desired, the app needs to be modified in order to interface with actual devices. Code for the most common devices is provided in comments in the app's source code, but other changes to the app's project may also be required.

UI flow

All examples given here use the Virtual Device Manager. Flow may be slightly different when an actual device is used.

Please note that a valid DC Gateway enpoint, username and password are required to process transactions.

- Upon start-up, the sample app enumerates all available devices in a drop-down box located in the upper left corner of the dialog (Figure 19, Figure 20).
- By pressing the <Connect> button, the sample app calls the Virtual Device Manager connect() method.
- The Virtual Device Manager displays the connect() dialog to simulate connecting to the device (Figure 21, Figure 22).
- As the sample app calls the Virtual Device Manager acceptCard() method, the Virtual Device Manager displays the associated dialog (Figure 23, Figure 24).
- Upon return, the sample app populates the Primary Account Number (PAN) and ExpDate fields from the simulated swipe with the Virtual Device Manager (Figure 25, Figure 26).
- By pressing the <Process Transaction> button, the sample app sends the transaction to the DC Gateway for processing.
- The result of the transaction is displayed in the Result, RespMSG and AuthCode fields (Figure 27, Figure 28).

Preliminary

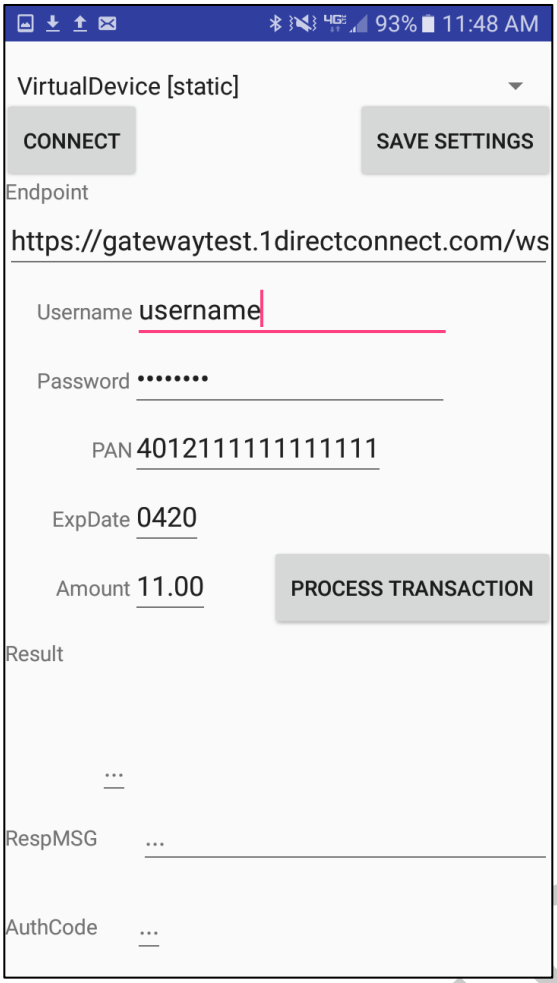

Figure 19: Android sample app start-up screen

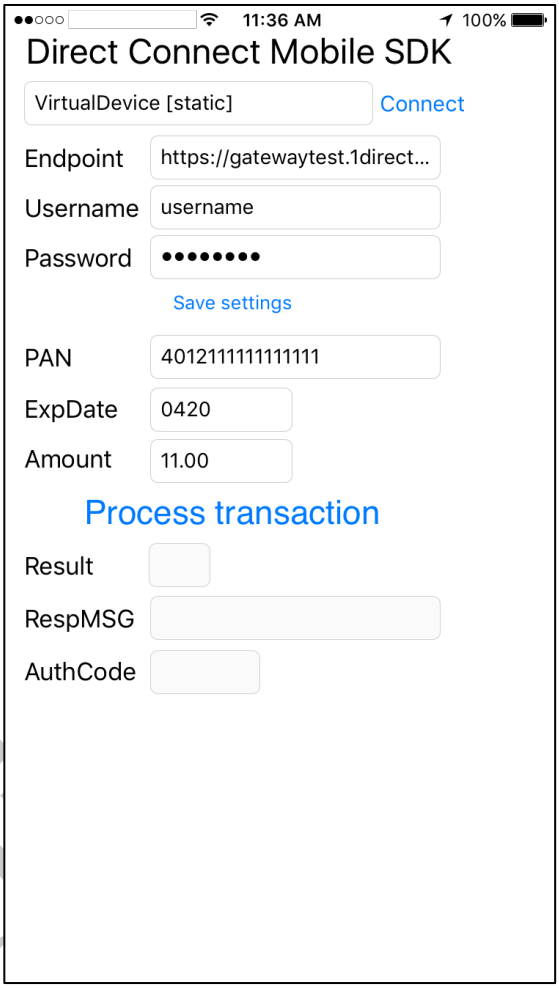

Figure 20: iOS sample app start-up screen

Preli

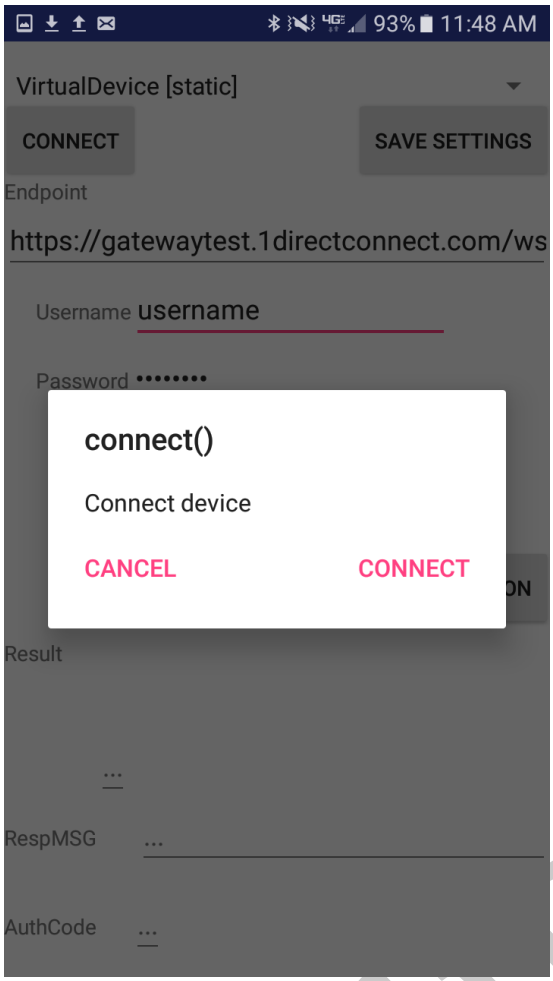

Figure 21: Android Virtual Device Manager connect() dialog

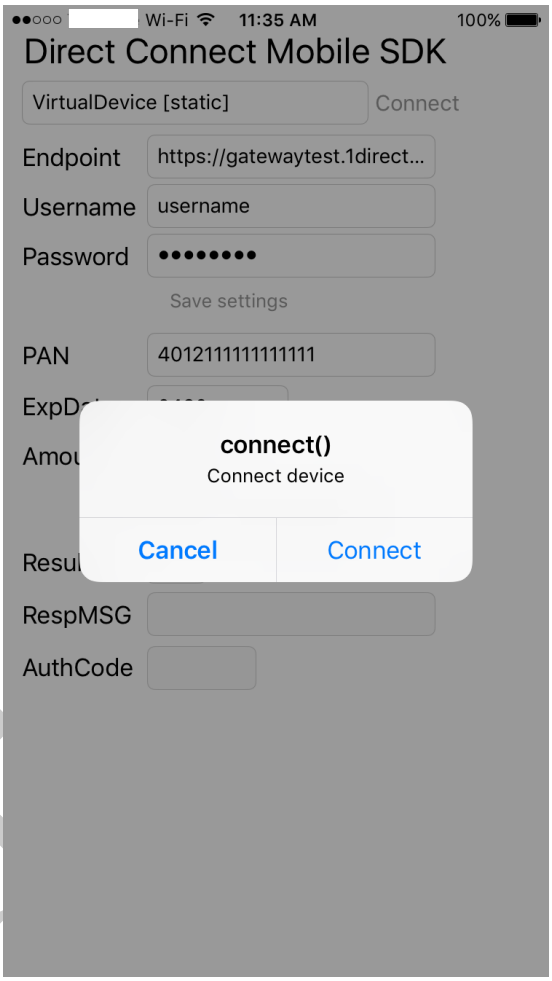

Figure 22: iOS Virtual Device Manager connect() dialog

Preli

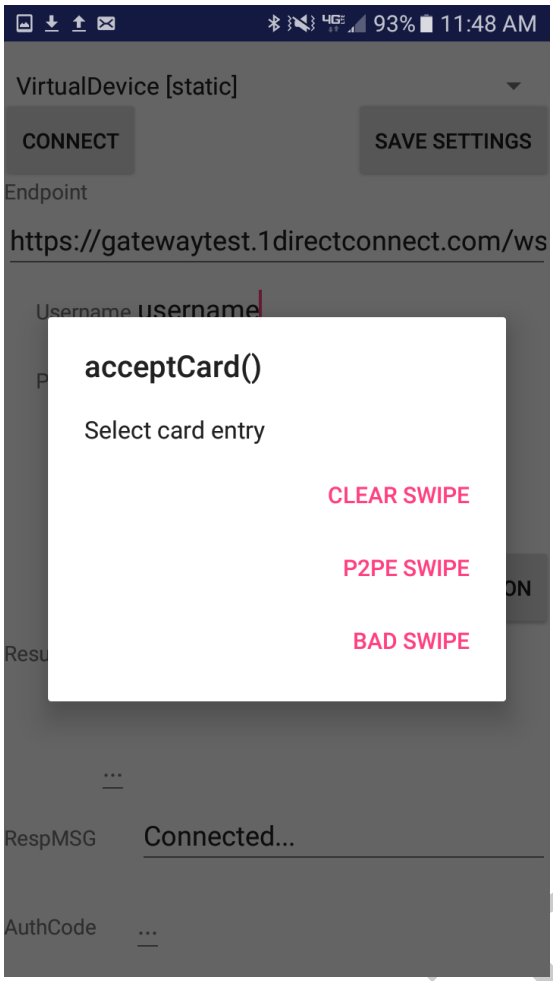

Figure 23: Android Virtual Device Manager acceptCard() dialog

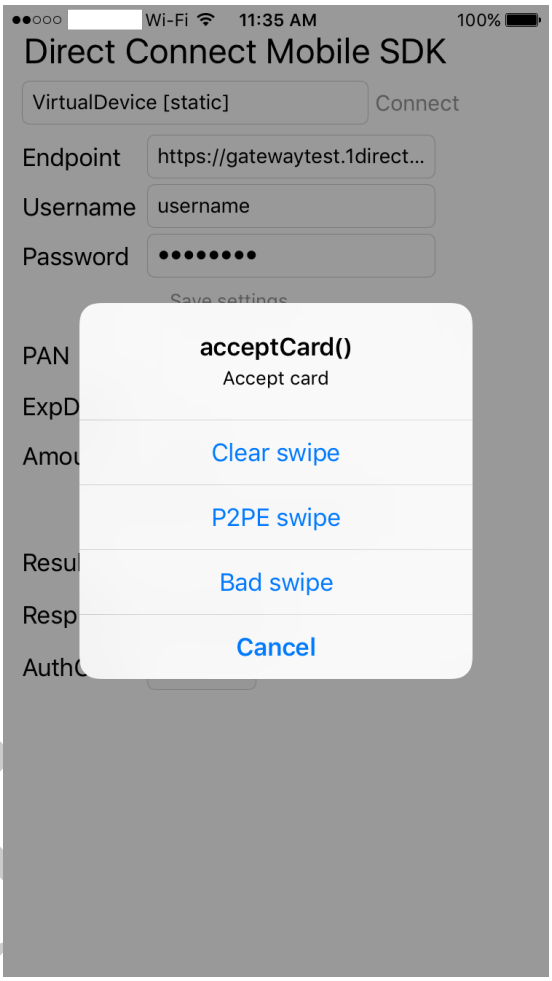

Figure 24: iOS Virtual Device Manager acceptCard() dialog

Prelim

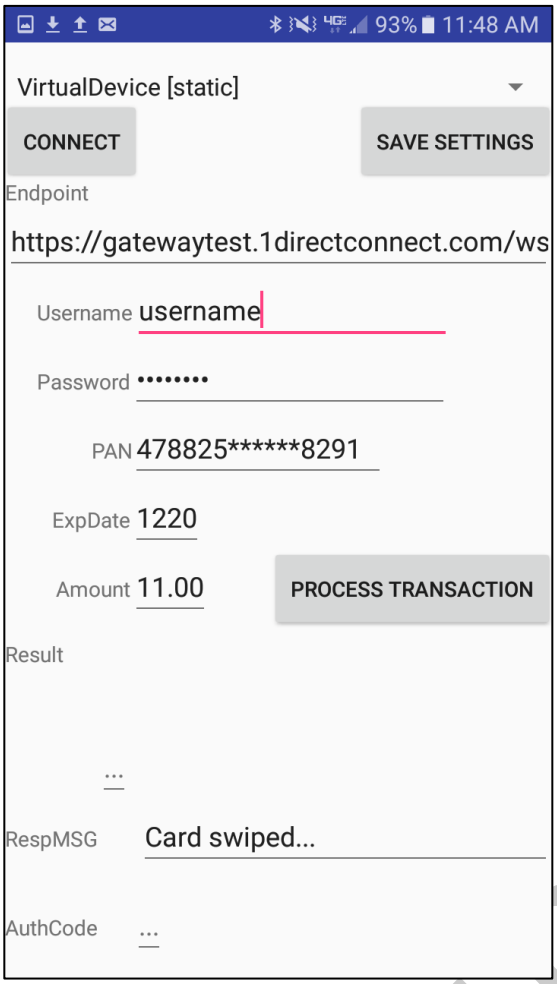

Figure 25: Android sample app with populated PAN field

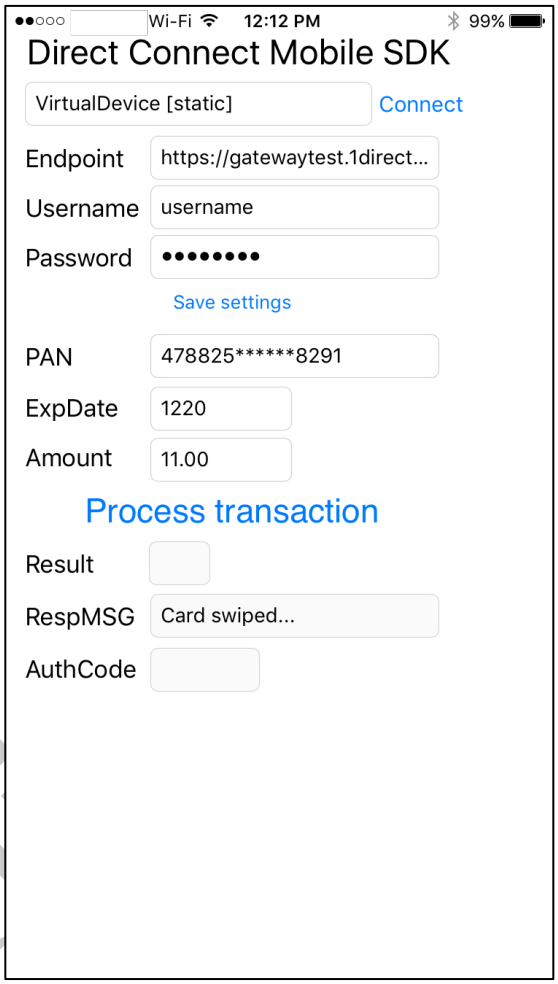

Figure 26: iOS sample app with populated PAN field

Preliminary

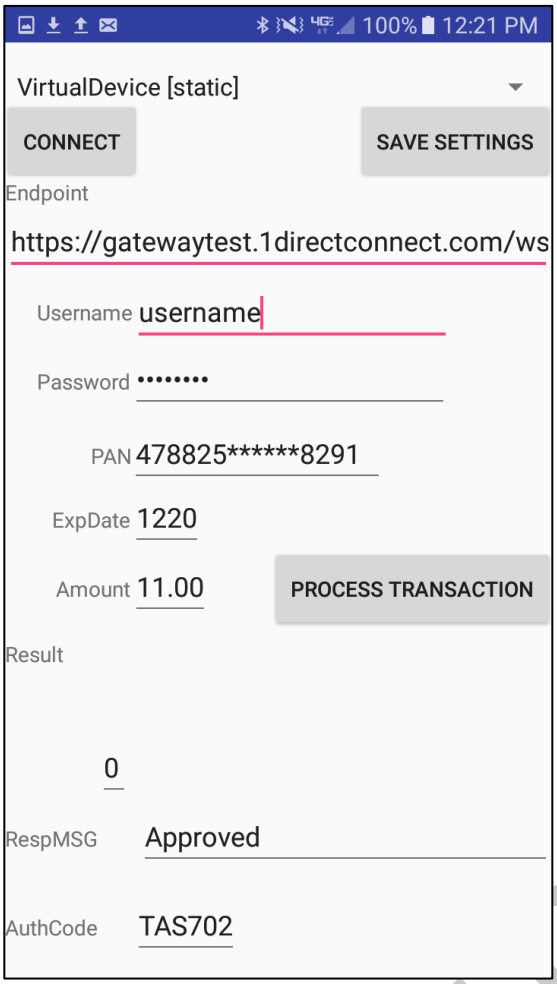

Figure 27: Android sample app transaction results

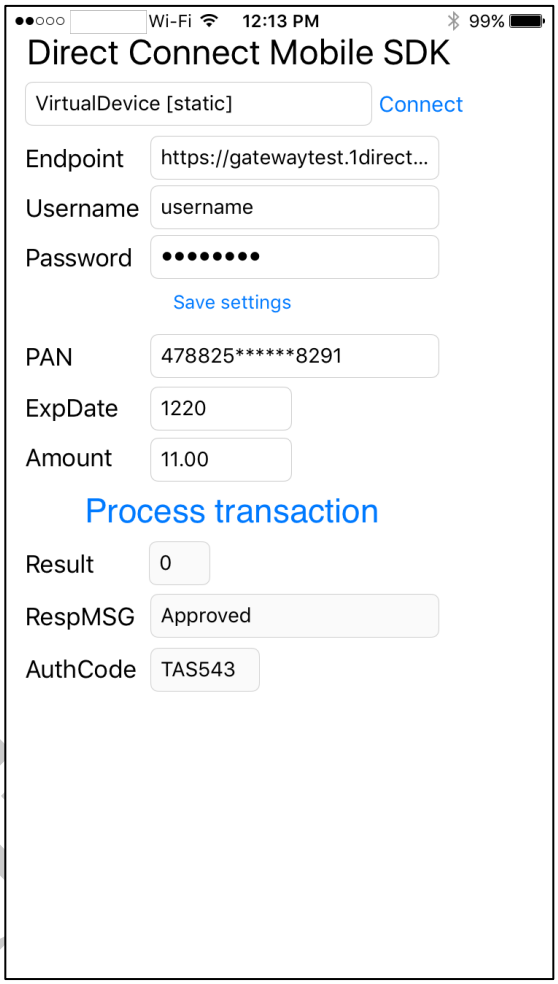

Figure 28: iOS sample app transaction results

Prelim