Quick Start Guide

Android version

Author: Wirecard Technologies GmbH

www.wirecard.com | info@wirecard.com

Contents

1 Introduction

This document describes how Accept SDK should be used by Android applications.

1.1 Audience

This document is intended for the developers who integrate Accept SDK into their Android applications.

1.2 Related Documents

1. Accept SDK Android Documentation - Full manual for Android SDK.
2. Public demo app source code repository: https://github.com/mposSVK/acceptSDK-Android/tree/master/demo/demo
3. Public SDK repository: https://github.com/mposSVK/acceptSDK-Android

In this document used as (RD1, RD2, RD3)

1.3 Revision History

SDK Version	Date	Name	Comments
1.5.7	06.12.2016	Marek H.	Added relations to GitHub, and GitHub demo source code files
1.4.9	08.01.2016	Patrik M.	Added 6. Step in the section 2: added loading of AID configuration after success initialization / login
1.0.0	06.08.2015	Damian K.	Fixed typos
1.0.0	31.07.2015	Damian K.	Initial version.

Copyright

Printed in Germany / European Union
Last updated: June 2015

Trademarks

2 Overview

Accept SDK for Android is an mPOS solution for Android devices that enables electronic payments through a range of terminals connected using Audio Jack or Bluetooth.

Currently the following terminals are supported:

1. Spire PosMate
2. Spire SPm2
3. BBPOS EMVSwiper

Accept SDK for Android is distributed as a set of AAR libraries or simple as linked dependency in gradle file (related documents) to GitHub repository. It consists of two or more aar files depending on which terminals are used for the payments. The set always contains the core library (*accept-sdk-android-acceptsdksource-release_x.x.x.aar*) and at least one aar-file more, as an extension for the core.
The extensions provide support for the terminals and allow the core to delegate the payment procedure to them. For example, if the application is going to use terminals for the payments, the SDK should have following aar files or links in gradle file:

- accept-sdk-android-acceptsdksource-release_x.x.x.aar
  https://github.com/mposSVK/acceptSDK-Android/blob/master/accept-sdk-android-acceptsdksource-release.aar

- and one (or both) of extensions files:
  i. accept-sdk-android-extension-spire-release_x.x.x.aar
     https://github.com/mposSVK/accept-android-extension-spire/blob/master/accept-sdk-android-extension-spire-release.aar
  ii. accept-sdk-android-extension-bbpos-release_x.x.x.aar
     https://github.com/mposSVK/accept-android-extension-bbpos/blob/master/accept-sdk-android-extension-bbpos-release.aar

2.1 Extension compatibility table

Current extensions-compatibility is written down in table. Current state is on github:

- acceptSDK-Android
- accept-android-extension-spire
- accept-android-extension-bbpos

3 SDK Integration

To integrate the Accept SDK please follow the steps below:

Note: integration using demo app source code repository requires basic knowledge about android project structure, and usage of product flavours.

This part of guide will explain implementation of Spire(Thyron) extension.

BBPos extension implementation is almost same.

- GitHub demo example

3.1 Using aar files

If your implementation will use direct ("in project" libs folder) link to aar files, you have to manually download aar files from GitHub repository every time if released some new version.
Copy Accept SDK AAR's files into your project directory and modify **build.grade** file:

```
apply plugin: 'com.android.application'

repositories {
    mavenCentral()
    flatDir {
        dirs 'aars'
    }
}

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.2.3'
    }
}

android {
    packagingOptions {
        exclude 'META-INF/notice.txt'
        exclude 'META-INF/license.txt'
        exclude 'META-INF/LICENSE'
        exclude 'META-INF/LICENSE.txt'
        exclude 'META-INF/NOTICE'
        exclude 'META-INF/NOTICE.txt'
    }
}

dependencies {
    compile (name: 'accept-sdk-android-acceptsdksource-1.4.7', ext: 'aar')
    compile (name: 'accept-sdk-android-thyronextension-1.4.7', ext: 'aar')
}
```

3.2 Using gradle and dependency

Simply copy gradle file from demo app for use it like maven gradle dependency:
https://github.com/mposSVK/acceptSDK-Android/blob/master/demo/demo/sample/build.gradle

If using maven it is important to specify maven repository link

```
repositories {
    maven { url "https://jitpack.io" }
    flatDir {
        dirs 'libs'
    }
}
```

And dependencies:

```
dependencies {
   compile 'com.fasterxml.jackson.core:jackson-databind:2.8.3'
   compile 'com.github.mposSVK:acceptSDK-Android:1.5.7'
   compile 'com.github.mposSVK:accept-android-extension-spire:1.5.5'
   //compile 'com.github.mposSVK:accept-android-extension-bbpos:1.5.2'
}
```

Note: *Extension compatibility is described here*

3.3 Modify AndroidManifest.xml file

to get the following permissions. This step is optional. Gradle builder merges the permissions from aar into application's manifest.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    ...
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    ...
</manifest>
```

Note: *Bluetooth permission required only for Spire extension.*

3.4 Initialize SDK

Add custom **Application** class to your project and add Accept initialisation procedure to Application class. This part can be written also in custom activity implementation. (Explained detail later)

```
public class Application extends android.app.Application {

    @Override
    public void onCreate() {
        super.onCreate();
        // Init loads the rest of configuration from config_for_accept.xml file.
        AcceptSDK.init(this, "{YOUR CLIENT ID}", "{YOUR CLIENT SECRET}",
                "{YOUR BACKEND INSTANCE URL}");
        AcceptSDK.loadExtensions(this, null);
    }
}
```

Note: *Please obtain Client ID/Secret from Accept support team.*
Everything is coded and commented in related documents.

- GitHub demo example

3.5 Create config file

Add Accept's configuration properties to res/values:

Note: We propose to **create config.xml** with this content in case of Spire(Thyron) extension used.

Every extension is independent, and implementer is able to choose which extension will be used by simply change **"extensions_list"** attribute in config file.
It is easy, compile your app with specified dependencies (gradle file) on extension and in config file simply say that you will use this extension.
SDK, to know which extension should be used is using reflection (wiki: “ability of a computer program to examine, introspect, and modify its own structure and behaviour at runtime”) for this feature we have to use reserved name “ThyronExtension”.
All attributes are self-describing.

```
<!--ThyronExtension is able to switch off contact less payments acceptance-->
<item name="acceptsdk_support_contactless" type="bool">true</item>

<item name="wl_default_terminal_type_use" type="string">ThyronExtension</item>

<!-- set list of supported extensions -->
<string-array name="extensions_list">
    <item>ThyronExtension</item>
</string-array>
```

Note: Again, everything is in case of BBPos implementation on GitHub.

- GitHub demo example

3.6 Payment flow

Let us start with flow explanation with introduce basic abstract class - controller.

```
abstract class PaymentFlowController

/**
 * discover devices and select one for using
 * @param context
 * @param delegate
 */
public abstract void discoverDevices(final Context context, final
DiscoverDelegate delegate);

/**
 * use for payment with device,
 * @param device Device - device identification
 * @param amount payment amount
 * @param currency payment currency
 * @param delegate delegate is interface PaymentFlowDelegate, responsible for
library status feedback and signature capturing , signature confirmation, errors
displaying
 */
public abstract void startPaymentFlow(final Device device, final long amount,
final Currency currency, final PaymentFlowDelegate delegate);

/**
 * cancel payment
 */
public abstract void cancelPaymentFlow();
```

And AcceptThyronPaymentFlowController as controller designed for Spire(Thyron) extension implementation.

3.7 Discover delegate, Device

Discover devices is first step in implementation. From integration, point is important to be familiar with basic interface and model used for data interchange. Using discover delegate you will get list of objects representing device identification used in next steps for payment.

Note: It is required to implement in *onDiscoveryDevices()* picker dialog to provide possibility to select terminal (from list of available) used for payment.

```
public interface DiscoverDelegate {
    void onDiscoveryError(DiscoveryError error, String technicalMessage);
    void onDiscoveredDevices(List<Device> devices);
}

public enum DiscoveryError {
    NO_BLUETOOTH_MODULE,
    NO_PERMISSION_TO_USE_BLUETOOTH,
    FAILED_TO_ENABLE_BLUETOOTH,
    NO_PERMISSION_TO_USE_MICROPHONE,
    AIRPLANE_MODE_ON,
    NOTHING_INSIDE_JACK,
    UNKNOWN
}
public class Device {
    public String displayName;
    public String id;
```

```
    public Device(){}
    public Device(String id,String displayName){
        this.id = id;
        this.displayName = displayName;
    }
}
```

3.8 Payment flow delegate

Second step is to make payment. From integration point of view is important to get familiar witch basic interface used for this functionality.

Payment flow delegate is basic object used for communication during payment. It is handling payment flow state updates, errors during payments, signature verification methods callbacks, etc.

```
public interface PaymentFlowDelegate {
    void onPaymentFlowUpdate(Update update);
    void onPaymentFlowError(Error error, String technicalDetails);
    void onPaymentSuccessful(Payment payment, String TC);
    void onSignatureRequested(SignatureRequest signatureRequest);
    void onSignatureConfirmationRequested(SignatureConfirmationRequest
signatureConfirmationRequest);
}

public enum Update {
    DATA_PROCESSING,
    LOADING,
    RESTARTING,
    ONLINE_DATA_PROCESSING,
    EMV_CONFIGURATION_LOAD,
    FIRMWARE_UPDATE,
    CONFIGURATION_UPDATE,
    BLUETOOTH_ENABLING,
    TRANSACTION_UPDATE,
    WAITING_FOR_INSERT,
    WAITING_FOR_SWIPE,
    WAITING_FOR_INSERT_OR_SWIPE,
    WAITING_FOR_INSERT_SWIPE_OR_TAP,
    WAITING_FOR_CARD_REMOVE,
    WAITING_FOR_AMOUNT_CONFIRMATION,
    WAITING_FOR_PINT_ENTRY,
    WRONG_SWIPE,
    UNKNOWN
}

public enum Error {
    NO_BLUETOOTH_MODULE,
    NO_PERMISSION_TO_USE_BLUETOOTH,
    BLUETOOTH_COMMUNICATION_FAILED,
    BLUETOOTH_PAIRING_LOST,
    DATA_PROCESSING_ERROR,
    ONLINE_PROCESSING_FAILED,
    ONLINE_PROCESSING_FAILED_CONNECTION_PROBLEM,
    TRANSACTION_UPDATE_FAILED,
    TRANSACTION_UPDATE_FAILED_CONNECTION_PROBLEM,
    EMV_CONFIGURATION_LOAD_FAILED,
    EMV_CONFIGURATION_LOAD_FAILED_CONNECTION_PROBLEM,
    EMV_CONFIGURATION_INVALID,
    CANCELED_ON_TERMINAL,
    UNKNOWN
}
```

3.9 Signature requirements callbacks

Signature is one of most used verification methods for verify card holder during payment.

SDK is saving (sending) signature image during payment into backend database. It is used two interfaces for handle this feature.

First one is for capturing signature. User should provide app to customer and request him to sign transaction directly on smartphone.

```
public interface SignatureRequest {
    void signatureEntered(byte[] signaturePNGBytes);
    void signatureCanceled();
}
```

Second interface is used for terminals without display for verify if signature is matching with one on backside of the payment card. e.g.: Spire terminals are solving this feature locally on terminal display, but BBPos requires implementing this callback to solve signature confirmation feature.

```
public interface SignatureConfirmationRequest {
    void signatureConfirmed();
    void signatureRejected();
}
```

Note: For Spire terminals, you can leave this handler empty and just display message about follow instructions on terminal display.

3.10 Terminal configuration

Online payment process is not so easy and it is normal to have some configuration. Our terminals are using configurations related to terminal capabilities, AID configurations, Merchant category code, Merchant identifiers, Terminal identifiers, Country specification, and e.q.
Usually is merchant requested to make one test payment every day morning, to test if connection is working, or if some configurations are not changed. Terminal configuration download and followed upload on terminal device is part of this first transaction. Usually it is followed by restarting terminal.

Mechanism of updating terminal is hidden in the call:

```
startPaymentFlow(final Device device, final long amount, final Currency currency,
final PaymentFlowDelegate delegate);
```

It is only once per day if first transaction is proceeded.

Note: transaction must be not completed to successful state. It is good enough to get state “Insert or swipe card” or “Present, insert or swipe card” in case of contact less supported.

Note: The terminal identification based configuration download is a part of merchant identification on backend. It needs to be configured with administrator. If you are getting some error messages / states before payment starts, please contact Accept support team, if it is not delivered together with the bundle.

4 Payment implementation

This part of document will mention about how to implement step after step all features related to payment.

4.1 Login

The payment procedure starts from a login operation. It initialises the Accept SDK and validates the account used for the payments:

```
final EditText usernameEditText = (EditText)findViewById(R.id.username);
final EditText passwordEditText = (EditText)findViewById(R.id.password);

final String username = usernameEditText.getText().toString();
final String password = passwordEditText.getText().toString();

AcceptSDK.login(username, password, new OnRequestFinishedListener<Object>() {
    @Override
    public void onRequestFinished(ApiResult apiResult, Object result) {
        if ( apiResult.isSuccess() ) {
            // login success.
            return;
        }
        // handle error: apiResult.getDescription());
    }
});
```

- GitHub demo example

After completes a successful login operation, the SDK gets the access token for further networking together with information about logged merchant. Basic merchant information can be obtained using the following methods:

```
final String merchantName = AcceptSDK.getMerchantName();
final String merchantEmail = AcceptSDK.getMerchantEmail();
final String merchantPhoneNumber = AcceptSDK.getMerchantPhoneNumber();
```

4.1.1 Session timeout

After successful login SDK gets the access token. It is representation of logged user. SDK will create session with (configurable) session timeout time. If this session validity time expired, user should login again. This event should be handled in app by implementation of Broadcast receiver.
Create pls broadcast receiver able to catch intent `AcceptSDKIntents.SESSION_TERMINATED.`

```
LocalBroadcastManager.getInstance(this).registerReceiver(receiver, new
IntentFilter(AcceptSDKIntents.SESSION_TERMINATED));

public void sendLogoutIntentAndGoLogin() {
    AcceptSDK.logout();
    Intent intent = new Intent(this, LoginActivity.class);
    intent.putExtra(BaseActivity.LOGOUT, true).addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    startActivity(intent);

    intent = new Intent(BaseActivity.INTENT);
    intent.putExtra(BaseActivity.INTENT_TYPE, BaseActivity.TYPE_LOGOUT);
    LocalBroadcastManager.getInstance(this).sendBroadcast(intent);
```

```
    }

    private class SessionTerminatedReceiver extends BroadcastReceiver {

        @Override
        public void onReceive(Context context, Intent intent) {
            Log.e("Session Timeout", "sending Log Out");
            sendLogoutIntentAndGoLogin();
        }
    }
```

Note: before every longer transaction is recommended to call session refresh

4.1.2 Session refresh

Session refresh is simple request which “says to backend I am alive, still working in app”

```
if (AcceptSDK.isLoggedIn()) {
    //use this if you will stay on same session, just app is working on longer
task
    AcceptSDK.sessionRefresh(new OnRequestFinishedListener<HashMap<String,
String>>() {
        @Override
        public void onRequestFinished(ApiResult apiResult, HashMap<String,
String> result) {
            if (!apiResult.isSuccess()) {
                sendLogoutIntentAndGoLogin();
            }
        }
    });
}
```

- GitHub demo example

When the SDK did a successful login, a new payment can be triggered using **PaymentFlowController**. The class is an abstraction layer for managing the payment procedure. Depending on which terminal is going to be used for the payment, a particular class of the payment flow controller needs to be created. For example, for the Spire/Thyron terminals, the SDK offers **AcceptThyronPaymentFlowController**.

4.2 Devices discovery

The device discovery procedure returns the list of currently available terminals. In case of Bluetooth terminals, it will return the list of paired Bluetooth devices. For Audio Jack devices, it will return the list of supported terminals.
A sample commented code for device discovery is presented below:

```
paymentFlowController.discoverDevices(this, new
PaymentFlowController.DiscoverDelegate() {

    @Override
    public void onDiscoveryError(final PaymentFlowController.DiscoveryError error,
final String technicalMessage) {
        //showTerminalDiscoveryError
    }

    @Override
    public void onDiscoveredDevices(final List<PaymentFlowController.Device>
```

```
devices) {
        if (devices.isEmpty()) {
             //showNoDevicesError
        }
        if ( devices.size() == 1 ) {
            currentDevice = devices.get(0);
        }
        //showTerminalChooser
            currentDevice = selected;
    }
});
```

- GitHub demo example

4.3 Handling payment

The payment procedure is driven by the SDK. It notifies the user about major events related to the payment using **PaymentFlowDelegate interface**. The same interface is also used to delegate some of the payment steps to the user of SDK, which are related to customer/merchant interaction. An example implementation of the PaymentFlowDelegate interface is presented below:

```
AcceptSDK.startPayment();// initialization of new payment in SDK

amountCurrency = Currency.getInstance(AcceptSDK.getCurrency());

Float tax;
if(AcceptSDK.getPrefTaxArray().isEmpty())//if not filled out use "0f"
    tax = 0f;
else tax = AcceptSDK.getPrefTaxArray().get(0);// taxes are defined on backend and
requested during communication..pls use only your "supported" values

//here is example how to add one payment item to basket
AcceptSDK.addPaymentItem(new PaymentItem(1, "", currentAmount, tax));
//for demonstration we are using only one item to be able to fully controll
amount from simple UI.

// and now we have to get amount in units from basket(with respect to taxes,
number of items....)
final long amountUnits =
AcceptSDK.getPaymentTotalAmount().scaleByPowerOfTen(amountCurrency.getDefaultFrac
tionDigits()).longValue();

//and finally start pay( with given device, pay specified units in chosen
currency)
```

- GitHub demo example

The beginning of the example shows how the Accept SDK must be configured before starting a new payment. The following sequence should always be performed before starting a new transaction:

1. AccpetSDK.startPayment() – it clears the data and starts a fresh payment.
2. AcceptSDK.addPaymentItem() – it adds a new item to customer's basket. The final amount is calculated from basket's content.

3. paymentFlowController.startPaymentFlow() – it runs the payment, establishes a session with the terminal and goes through the payment.

4.4 PaymentFlowDelegate

Is basic representation of interface used for whole payment process.
Simple explanation and example:

```
paymentFlowController.startPaymentFlow(device, amountUnits, amountCurrency,
        new PaymentFlowController.PaymentFlowDelegate() {

    @Override
    public void onPaymentFlowUpdate(PaymentFlowController.Update update) {
        // Callback notifies about the progress of payment.
    }

    @Override
    public void onPaymentFlowError(PaymentFlowController.Error error,
                                   String technicalDetails) {
        // Callback notifies about an error occurred during the payment.
    }

    @Override
    public void onPaymentSuccessful(Payment payment, String TC) {
        // Callback notifies about a successful end of the payment.
    }

    @Override
    public void onSignatureRequested(PaymentFlowController.SignatureRequest
signatureRequest) {
        // Callback delegates the payment signing to the application.
        // Customer needs to draw a signature and then application needs to call
signatureEntered() or signatureCanceled().
    }

    @Override
    public void onSignatureConfirmationRequested(
            PaymentFlowController.SignatureConfirmationRequest
signatureConfirmationRequest) {
        // Callback delegates signature confirmation to the application.
        // The app needs to inform merchant he needs to confirm the signature
using terminal's keypad.Some terminals are handling this feature using terminal
display and buttons.
    }
});
```

4.4.1 onPaymentFlowUpdate

callback method informs about what is happening with the SDK. The method can be called very often comparing to the others. Usually, the implementation of the method should update a label informing merchant and customer about the state of payment. The list of updates is available at **Appendix A.**

4.4.2 onPaymentFlowError

callback method is invoked when an error occurred during the payment. The list of errors is available at **Appendix B.** After this callback no more methods will be called.

4.4.3 onPaymentSuccessful

callback method is invoked when the payment is accepted. After this callback no more methods will be called.

4.4.4 onSignatureRequested

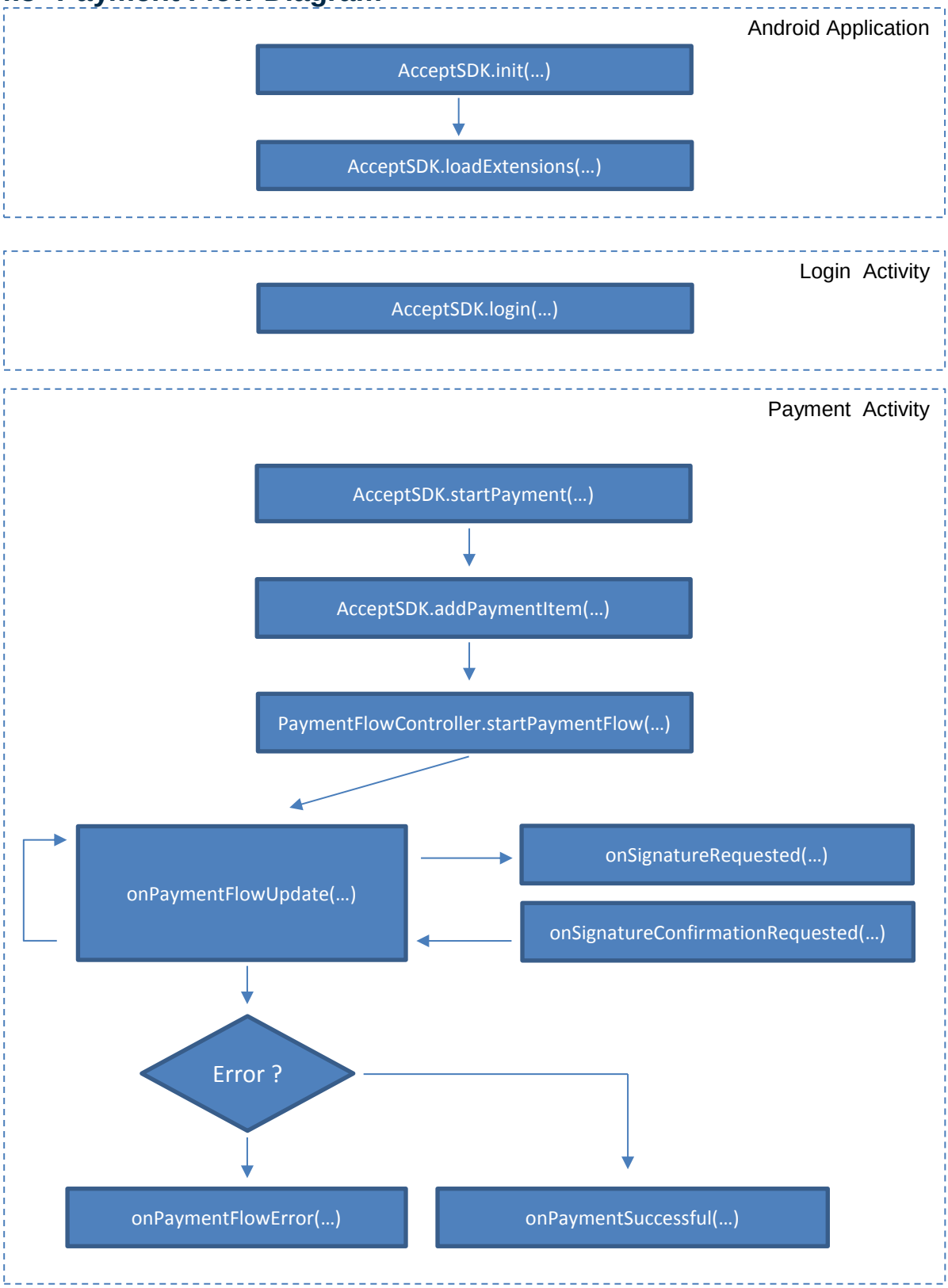
callback method is invoked when payment procedure requires customer's signature. The app should display a drawing canvas and collect the signature image file from what user drawn. When the signature is ready the SDK is notified by **SignatureRequest:: signatureEntered(byte[] pngBytes).** This callback is optional depending which terminal and card are used.

4.4.5 onSignatureConfirmationRequested

callback method is invoked when the SDK asks to display a message to a merchant that he needs to confirm signature on terminal's keypad. The application should display a signature drawn by the customer and ask merchant to confirm it using terminal's keypad. This callback is optional depending which terminal and card are used.

- **Note:** all implementations example you can find on GitHub demo example

4.5 Payment Flow Diagram



4.6 Firmware update (Spire only)

This is only Spire (Thyron) related.
SDK supports firmware update feature for Spire terminals only. It is required to download firmware configuration files separately.

This feature contains two steps:

- First step is *AcceptSDK.fetchFirmwareVersionInfo()* is only about fetch information's about new firmware availability. SDK remembers last updated version of firmware locally, and if its different app should show "Firmware update" activity, which can be easy implemented by copy source code from GitHub demo app, you can find it in MainMenuActivity in Demo app.

  Note: simply look for *class FirmwareCheckTask extends AsyncTask*

- Second step is to download files using SDK support:

```
TerminalInfo.downloadSaveAndExtractZipFile(FirmwareUpdateActivity.this,
                    firmwareNumberAndUrl.getFwUrl());
```

  This method should be started as async task, because is time consuming. After successful download and extracting all files to local memory it is needed just reconnect to terminal and during connection SDK will automatically update this files.
  Detailed implementation you can find on GitHub demoapp *FirmwareUpdateActivity*

  Note: this activity is using simplest way how to connect and start communication with Spire terminal. During start of communication is SDK simply comparing firmware numbers and start upload as additional step. After update it can happen that restart will be needed.

5 Appendix: A

Payment flow updates class details (**PaymentFlowController.Update**):

CONFIGURATION_UPDATE	SDK updates terminal's configuration remotely.
FIRMWARE_UPDATE	SDK updates terminal's firmware remotely.
LOADING	SDK is processing please wait.
RESTARTING	SDK restart the terminal an waits for the reboot.
ONLINE_DATA_PROCESSING	SDK performs online processing of the payment.
EMV_CONFIGURATION_LOAD	SDK load properties required for CHIP transactions.
DATA_PROCESSING	SDK notifies about the communication between the card and terminal.
WAITING_FOR_CARD_REMOVE	SDK get a information from the terminal it waits for merchant to remove the card. An application should display a text: "PLEASE REMOVE CARD".
WAITING_FOR_INSERT	SDK get a information from the terminal it waits for merchant to insert the card. An application should display a text: "PLEASE INSERT CARD".
WAITING_FOR_INSERT_OR_SWIPE	SDK get a information from the terminal it waits for merchant to insert or swipe the card. An application should display a text: "PLEASE INSERT OR SWIPE CARD".
WAITING_FOR_PINT_ENTRY	SDK get a information from the terminal it waits for the PIN entry. An application should display a text: "PLEASE ENTER PIN".
WAITING_FOR_SWIPE	SDK get a information from the terminal it waits for merchant to swipe the card. An application should display a text: "PLEASE SWIPE CARD".
WAITING_FOR_AMOUNT_CONFIRMATION	SDK get a information from the terminal it waits for customer to confirm the amount. An application should display a text: "PLEASE CONFIRM AMOUNT".
TRANSACTION_UPDATE	SDK performs another online processing of the transaction to finally commit it.
WRONG SWIPE	SDK inform about wrong swipe

- GitHub demo example of implementtion

6 Appendix: B

Payment error class details (**PaymentFlowController.Error**):

Note: Not all erros can occur while using particular extension. For example, Audio Jack terminals will never report **NO_BLUETOOTH_MODULE** error.

NO_BLUETOOTH_MODULE	The device is not equipped with Bluetooth module.
NO_PERMISSION_TO_USE_BLUETOOTH	The application does not have rights to use the Bluetooth module. No Bluetooth permissions in AndroidManifest.xml file.
BLUETOOTH_COMMUNICATION_FAILED	An IO error occurred during data transmission to or from the terminal.
BLUETOOTH_PAIRING_LOST	The Bluetooth device is not paired anymore.
DATA_PROCESSING_ERROR	Transaction processing by the card has failed.
ONLINE_PROCESSING_FAILED	Transaction rejected by backend.
ONLINE_PROCESSING_FAILED _CONNECTION_PROBLEM	No internet connection.
TRANSACTION_UPDATE_FAILED	Transaction finalization failed.
TRANSACTION_UPDATE_FAILED _CONNECTION_PROBLEM	No internet connection.
EMV_CONFIGURATION_LOAD_FAILED	Failed to load the EMV configuration.
EMV_CONFIGURATION_LOAD_FAILED _CONNECTION_PROBLEM	No internet connection.
EMV_CONFIGURATION_INVALID	The validation of EMV configuration failed.
CANCELED_ON_TERMINAL	The payment was cancelled using a button from terminals' keypad.
UNKNOWN	SDK experienced an error which is not handled yet.