Add-on PCL for Android 1.30.00

Integration Guide

ICO-OPE-00520-V41
Restricted

2023/04/24

www.ingenico.com

28/32, boulevard de Grenelle, 75015 Paris - France / (T) +33 (0)1 58 01 80 00 / (F) +33 (0)1 58 01 91 35

Ingenico – S.A. au capital de 53 086 309 € / 317 218 758 RCS PARIS

Contents

1 Introduction

This document will guide you to develop Android applications using “Add-on PCL for Android”.

Using this Add-on, you will be able to:

- launch a payment transaction,
- exchange messages between an Ingenico devices and an iOS device,
- remotely manage the payment terminal,
- print text and bitmap images on the terminal printer,
- capture a signature on the iOS device
- scan bar code using companion bar code barcode reader if available

Refer to the Release Note to have details about features that have been removed or added compared to the previous versions.

1_1 Acronyms and terminology

Acronym	Description
PCL	Payment Communication Layer
Terminal	Ingenico device that is used to manage the payment. It can be any of Ingenico device from running under Telium2 or above operating system
mobile device	Either a tablet or a smartphone running under Android Operating System
IP	Internet Protocol, just a technical name for your local network (either Wi-Fi or wired)

 This symbol indicates an important Warning.

 This symbol indicates a piece of advice.

2 Installation

2_1 Add-on content

Before proceeding with the installation steps, you must extract the content of the Add-on on your computer in a %Add-on_Dir%.
The “Add-on PCL for Android” contains all the binaries needed to ensure the communication between an Android device and a Telium device, in order to perform payment transactions, to manage terminal and to use terminal peripherals (printer, barcode reader)…
It also contains all the files and documentation to allow the development of Android applications that interact with Ingenico Companion.

2_2 Prerequisites

Please refer to the Release Note document for hardware and software prerequisites on both Android and Telium side.

2_3 Installing the Add-on PCL on the Android device

Ingenico provides 3 Android applications to help getting started with one’s own application:

File name	Description
PclTestApp.apk	Provide unit test functions of all the PCL features.
SpicesShop.apk	Example of how the sale application of a spice shop could look like.
PCLFileSharingSample.apk	Sample application demonstrating the PCL File Sharing features.

To install the “Add-on PCL for Android”, follow the below steps:
- Go the directory %Add-on_Dir%\Packages
- Copy the .apk files contained in this folder on the Android device

For more details please read the document “Getting started with PCL for Android” provided in this add-on.

3 Setting up the development environment

3_1 Prerequisites

To develop an Android application, the following are needed:

- a Java environment
- The Android SDK and development tools that can be downloaded here. Read carefully the "SDK Readme.txt" file after the execution of the SDK installation process as it contains important information related to the next installation steps

In addition, it is possible to install an Android emulator to test the application on the computer instead of testing it on the mobile device.
It is also strongly recommended to use an IDE (such as Android Studio).

3_2 Creating a new application

Refer to your IDE user guide for information about creating a new application. Note that depending on the IDE you are using, you may have an option to "Create a new Android application".
Once done, you need to include the PCL files needed by the project.

1. Copy `%Add-on_Dir%\SDK\Java Libraries\PclServiceLib.jar` and `PclUtilities.jar` in the `libs/` directory of your project
2. Copy native libraries (`%Add-on_Dir%\SDK\Native Libraries\<arch>\*.so` files) in `libs/<arch>/` directory of your project
3. Add a reference to `PclServiceLib.jar` and `PclUtilities.jar` in your Build Path
4. Add PclService and BluetoothService in your application manifest:
```
<service android:name="com.ingenico.pclservice.PclService" />
<service android:name="com.ingenico.pclservice.BluetoothService" />
```
5. Add the following permissions in your application manifest:
```
android.permission.INTERNET
android.permission.WRITE_EXTERNAL_STORAGE
android.permission.ACCESS_NETWORK_STATE
android.permission.BLUETOOTH
android.permission.BLUETOOTH_ADMIN
android.permission.ACCESS_WIFI_STATE
android.permission.READ_PHONE_STATE
```

PclUtilities.jar must now be included in your project because PclServiceLib.jar requires this library.

android.permission.READ_PHONE_STATE permission is no more required in applications embedding PclServiceLib.jar. However it is needed if you use Telium SPMCI API PDA_get_comm_periph_state to retrieve GPRS SIM number.

You can use `PclServiceLib.aar` and `PclUtilities.aar` instead of jar and native libraries.

3_3 Binding to the PCLService

3_3_1 Overview

The communication between an Android application and an Ingenico Companion is done through the PclService library that is delivered in the “Add-on PCL for Android”.

The Android application that wants to interact with a Companion must be bound to PclService.

For more details about Bound Services please read Android developer documentation here.

3_3_2 Binding steps:

1. Declare an instance of the `PclService` interface.
2. Implement `ServiceConnection`.
3. In your implementation of onServiceConnected(), you will receive an IBinder instance (called service). Cast the returned parameter to PclBinder type and get PclService instance with getService(). Old extension for Binder still exists and works.
4. To connect, call Context.bindService(), passing in your ServiceConnection implementation.
5. Call the methods defined on IPclService interface.
6. To disconnect, call Context.unbindService() with the instance of your interface.

```
public class YourActivity extends Activity {
  // Declare PclService interface
  protected PclService mPclService = null;
  // Declare Serviceconnection
  private PclServiceConnection mServiceConnection;

  // Implement ServiceConnection
  class PclServiceConnection implements ServiceConnection
  {
    public void onServiceConnected(ComponentName className,
        IBinder boundService )
    {
      // We've bound to PclService, cast the IBinder and get PclService
instance
      PclBinder binder = (PclBinder) boundService;
      mPclService = (PclService) binder.getService();
    }

    public void onServiceDisconnected(ComponentName className)
    {
      mPclService = null;
    }
  };

  // You can call this method in onCreate for instance
  private void initService()
  {
    mServiceConnection = new PclServiceConnection();
    Intent intent = new Intent(this, PclService.class);
    bindService(intent, mServiceConnection, Context.BIND_AUTO_CREATE);
```

```
  }

  // You can call this method in onDestroy for instance
  private void releaseService()
  {
    unbindService(mServiceConnection);
  }
};
```

3_4 Android 8/9 behavior changes

Android 8.0 (API level 26) and higher impose limitations on what apps can do while running in the background, especially, while an app is idle, there are limits to its use of background services.

By default, these restrictions were only applying to apps that targeted Android 8.0 (API level 26) or higher.
For more details, please see https://developer.android.com/about/versions/oreo/background#services .

But Android 9 (API level 28) introduces new features to improve device power management.
When battery saver is turned on, the system places restrictions on all apps.
Depending on manufacturers, background execution limits could apply to all apps when battery saver is turned on, regardless of their target API level, refer to:
https://developer.android.com/about/versions/pie/power#battery-saver .

PCL service is an Android service that runs by default in the background, so it is affected by these limitations.

So, depending on Android running version, on application target API and on Android system state, PCL service can be stopped at any time when the application is in background.
This can be observed in the logs after 20, 2 minutes of idle as shown in the below logcat trace:

```
ActivityManager: Stopping service due to app idle: u0a284 -20m48s318ms
com.ingenico.pcltestappwithlib/com.ingenico.pclservice.PclService
ActivityManager: Stopping service due to app idle: u0a284 -2m9s359ms
com.ingenico.pcltestappwithlib/com.ingenico.pclservice.PclService
```

When this occurs the connection with the terminal is lost and never recovered because PCL service doesn't restart itself.

Starting with "Add-on PCL for Android 1.19.01", PCL service use startForeground on Android 9 and later and send foreground notification on channel "PCL".
To manage the foreground notification by the application, foreground mode can be disabled
This is done by adding the following extras in the Intent at service starting:

```
i.putExtra("FOREGROUND", false);
```

In this case, to avoid PCL stop in background, a foreground service must bind PCL service. When calling startForeground, the application will have to define a notification that will appear in the status bar.
In the latter case, the application would also need to restart PCL service when the application comes back to foreground.

Note that applications targeting API 28 must also declare in their manifest the use of a new permission FOREGROUND_SERVICE.

3_5 Activating a terminal

To activate a terminal you will need to add PclUtilities.jar library in libs folder of your project and add a reference to this library in the build path.

Depending on the type of Terminal you want to manage, several commands must be added.

In all cases, the first step to perform is to instantiate a PclUtilities object as the Terminal management is managed by this class.

Instantiate PclUtilities with the context of your application, the package name to use and the name of the file that contains the Bluetooth pairing information:

```
   PclUtilities mPclUtil = new PclUtilities(this, "com.ingenico.pcltest",
"pairing_addr.txt");
```

The sections below describe the main function of `PclUtilities`. Please refer to the Javadoc for more details about the `PclUtilities` classes and methods.

3_5_1 Using Bluetooth Terminals

To be able to use a Bluetooth Terminal, it must be paired (from a Bluetooth perspective) with the Android device.
Detailed information about the pairing process can be found in the “Getting started with PCL for Android” document.

PclUtilities class provides 2 APIs to use Bluetooth Terminals:

`Set<BluetoothCompanion> GetPairedCompanions()`: to retrieve the list of paired companions

`int ActivateCompanion(String sBtAddress)`: to activate the Terminal with the given Bluetooth address. This method stores the Bluetooth address of the activated companion in the internal storage of the application specified by package name used in PclUtilities constructor. Alternatively, you can use the `activate()` method of the `BluetoothCompanion` class.

You can find an implementation sample of these features in the PclTestApp source code (in the WelcomeActivity.java and TestListActivity.java files).

PclUtilities uses some hardcoded Bluetooth address ranges for Ingenico terminals.
Starting with “Add-on PCL for Android 1.10.00” you can add other Bluetooth address ranges with `AddAuthorizedBTAddr(String pAddr)` method.

Starting with "Add-on PCL for Android 1.18.00”, PclTestApp sample provide activity code and resources for easypairing. With PclUtilities API isIngenicoBtDevice() and Bluetooth functions, the activity has capacity to list discoverable terminals and run pairing to selected one.
In order to use it, add EasyPairingActivity.java with easypairing.xml, easypairing_listrow.xml and easypairing_string.xml in your project.
In manifest, add permission to manage Bluetooth discovering and pairing:

```
<uses-permission android:name=”android.permission.BLUETOOTH”/>
<uses-permission android:name=”android.permission.BLUETOOTH_ADMIN”/>
<uses-permission android:name=”android.permission.ACCESS_COARSE_LOCATION”/>
```

Then use the activity as usual (added in manifest and launched by intent).

3_5_2 Using USB Terminals

To connect to Telium terminal through USB, the Android device must be in USB host mode: it will power the bus and enumerate the connected USB devices.
USB host mode is only supported in Android 3.1 and higher.

Using USB host mode imposes the following requirements:

- Because not all Android-powered devices are guaranteed to support the USB host APIs, include a <uses-feature> element that declares that your application uses the android.hardware.usb.host feature.
- Set the minimum SDK of the application to API Level 12 or higher. The USB host APIs are not present on earlier API levels.

Before communicating with the USB device, your application must have the permission from your users. You can explicitly ask for permission when your application wants to communicate with a USB device but this permission will be revoked when the USB device is unplugged.
So the recommended way to obtain a permanent permission to communicate with a USB device is to use an intent filter in your application's manifest.

Your application must obtain the permission to communicate with a USB device.
Add an intent filter in one of the activity (the launcher activity for example) to filter for the `android.hardware.usb.action.USB_DEVICE_ATTACHED` intent. Along with this intent filter, you need to specify a resource file that specifies properties of the USB device, such as product and vendor ID. When a user connects a device that matches your device filter, the system presents him with a dialog that asks if he wants to start your application. If the user accepts, your application automatically has permission to access the device until the device is disconnected. If the user accepts to set your application as the default application to use the USB device, the permission will be granted forever (unless the user resets the application parameters or uninstalls the application).

The following example shows how to declare the intent filter (use the singleTask launchMode to avoid launching the activity each time a USB device is attached):

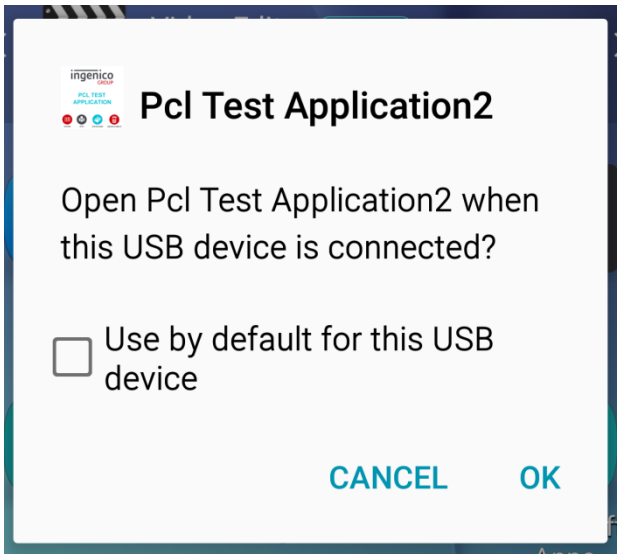
```
<activity …
    android:launchMode=”singleTask” >
…
    <intent-filter>
        <action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
    </intent-filter>
    <meta-data android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
android:resource="@xml/usb_device_filter" />
</activity>
```

The following example shows how to declare the corresponding resource file that specifies the USB devices that you're interested in (in this case, a Telium generic vendor id and product id):

```
<?xml version="1.0" encoding="utf-8"?>

<resources>
    <usb-device vendor-id="1947" product-id="40" />
</resources>
```

As an example, if you plug a USB companion, a popup requesting the permission will be displayed.
If the user checks “Use by default for this USB device” and then clicks OK, the permission will be granted forever the pop-up will no longer appear.



For more details about Android USB host mode, please refer to the Android developer reference here..

The PclUtilities class provides the following APIs to deal with USB Terminals:

`Set<UsbCompanion> getUsbCcompanions()`: to retrieve the list of Terminals connected through USB

`int activateUsbCompanion(String name)`: to activate the USB Terminals whose name is given in parameter. The name is defined as <terminal type>-<serial number>, eg. iCT250-01234567. The name is a member of the `UsbCompanion` class so you can get it with the `getUsbCompanions()` method. Alternatively, you can use the `activate()` method of the `UsbCompanion` class.

PclUtilities uses some hardcoded USB VID/PID for Ingenico terminals.
Starting with “Add-on PCL for Android 1.10.00” you can add other VID/PID with `AddAuthorizedVidPid(int vid, int pid)` method.

3_5_3 Using IP Terminals

One of the new features of the Add-on PCL for Android 1.10.00 is the ability to use a Terminal connected on the same local network. The Terminal can be connected either using the Wi-Fi or using an Ethernet port on a Wi-Fi hub.

The PclUtilities class provides the following APIs to deal with IP Terminals:

`Set<IpTerminal> getIPTerminals()` to you to retrieve the list of Terminal connected to the same local network as the Android device. Note that this function must be called in a separate Thread. The correct way to call it is as described below.

`activateIpTerminal(PclUtilities.IpTerminal terminal)` allows you to activate an Terminal located on the same local network. The terminal parameter is an object retrieved with a previous call to the `getIPTerminals()` function.

The connection between the Terminal and the Device can be secured using SSL.
To use SSL:

- Put the .p12 file corresponding to the Device certificate in the chosen Device directory. Take care of permissions needed by the Android application access at the location.
- Put the corresponding root key in a SPMCI_CA.CRT file to be uploaded in the /HOST directory of the Terminal. Take care of matching case for file name.
- Specify the certificate parameters in the sslObject that will be used as a parameter of the PclService startup (see below).

For mutual authentication:

- Put the same root key in file as SPMCI_CA.CRT in the chosen Device directory as .p12
- Put the corresponding Terminal certificate in SPMCI_CL.PEM file and corresponding Terminal private key in SPMCI_CL.KEY file to be uploaded in the /HOST directory of the Terminal. Take care of matching case for files names.
- Specify the certificate parameters in the sslObject that will be used as a parameter of the PclService startup (see below).

3_5_4 Using RS232 Terminals

One of the new features of the Add-on PCL for Android 1.12.00 is the ability to connect to Telium terminal through RS232.
The PclUtilities class provides the following APIs to deal with RS232 Terminals:

`Set<String> getSerialPortDevices()`: to retrieve the list of serial port on android device.

`SerialPortCompanion getSerialPortCompanion(String device)`: to retrieve the Terminal connected whose serial port device is given in parameter.

`int activate SerialPortCompanion(String device, String name)`: to activate the RS232 Terminals whose name and serial port device are given in parameter. The name is defined as <terminal type>-<serial number>, eg. iCT250-01234567. The serial port device is defined as <path to device>, eg. /dev/ttyS0. The name and serial port device is a member of the `SerialPortCompanion` class so you can get it with the `getSerialPortCompanion()` method. Alternatively, you can use the `activate()` method of the `SerialPortCompanion` class.

Don't use the PclUtilities API (`getSerialPortCompanion,…`) when the PCL link is established.

3_5_5 Starting PCLService

You need to specify additional parameters to the PclService when starting it.

In case of Bluetooth Terminals, you need to specify where to find the Bluetooth address of the activated companion. So you will need to add some extras in the Intent used to start or bind the service to provide the same package and file names:

```
private void startPclService() {
  if (!mServiceStarted)
  {
    Intent i = new Intent(this, PclService.class);
    i.putExtra("PACKAGE_NAME", "com.ingenico.pcltestapp");
    i.putExtra("FILE_NAME", "pairing_addr.txt");
```

```
    if (getApplicationContext().startService(i) != null)
      mServiceStarted = true;
  }
}
```

In case of IP Terminals, you need to specify the Terminal that has been selected and some SSL parameters (if SSL is used). This is done by adding the following extras in the Intent:

```
i.putExtra("IS_SSL", sslActivated);
    where sslActivated is a variable specified whether the connection will be done in SSL or not

i.putExtra("IS_IP_TERMINAL", isTerminalIPActivated);
   where isTerminalIPActivated specifies whether we are going to use IP Terminals or not

i.putExtra("SSL_KEYSTORE", sslKeyStore);
   where sslKeyStore is a SslObject object (as defined in PclUtilities library) that is used to specify the File object and password of the private certificate to use for the SSL communication

i.putExtra("SSL_TRUSTSTORE", sslTrustStore);
   where sslTrustStore is a SslObject object (as defined in PclUtilities library) that is used to specify the File object and password of the public certificate to use for the SSL communication
```

If SSL_TRUSTSTORE is added, PclService will try mutual authentication. In the other case, SSL will be in simple authentication.

To manage certificate invalidity, an Android application can check certificate issue in either way:
- Call the PclService isSslCertificateKo method on a regular basis
- Implement a broadcast receiver that will listen for `"com.ingenico.pclservice.intent.action.SSL_CERTIFICATE_READ"` and get the extra string `"ssl_certificate"` when the broadcast is received. The string value can be `"VALID"` or `"INVALID"`.

3_6 Managing logs

3_6_1 Disabling Logs

Starting with “Add-on PCL for Android 1.07.00”, you can disable and enable the logs in PclService. There are two ways to achieve this:
- When starting or binding to PclService
- Through a PclService API

To disable the logs when starting or binding to PclService you need to add an extra to the Intent used to start or bind to the service:

```
private void startPclService() {
  if (!mServiceStarted)
  {
```

```
    Intent i = new Intent(this, PclService.class);
    i.putExtra("ENABLE_LOG", false); // Disable logs

    if (getApplicationContext().startService(i) != null)
      mServiceStarted = true;
  }
}
```

To enable or disable the logs dynamically once PclService has been started, you need to call the `enableDebugLog(boolean)` method.

Take care of permissions managed by the application to write logs files at the targeted directory

3_6_2 Logs Files

Starting with “Add-on PCL for Android 1.10.00”, the logs are also available in 4 text files located in the folder Android/data/<application package name>/files, PclServiceLog.txt, PclUtilitiesLog.txt, PclServiceLog0.txt and PclUtilitiesLog0.txt.
The files size is limited to 1MB and logging uses log rotation between Log and Log0 files (PclServiceLog.txt and PclUtilitiesLog.txt always contain the most recent logs).

Starting with “Add-on PCL for Android 1.18.00”, the logs can be obtained from android logcat. The logs are stocked into 5 text files:
PclServiceLog.txt, PclServiceLog.txt.1 … PclServiceLog.txt.4 (PclServiceLog.txt always contains the most recent logs).
On “Add-on PCL for Android 1.18.01” and later, you need to activate this feature manually.
Log file management can be configured to switch between the two ways. When starting or binding to PclService you need to add an extra to the Intent used to start or bind the service:

```
private void startPclService() {
  if (!mServiceStarted)
  {
    Intent i = new Intent(this, PclService.class);
    i.putExtra("LOGGER_LOGCAT", true); // write logs files using logcat

    if (getApplicationContext().startService(i) != null)
      mServiceStarted = true;
  }
}
```

Using android logcat management is not guaranteed on Android 6 and earlier.

Starting with "Add-on PCL for Android 1.18.00" the logs can be located in folder specified by the application. When starting or binding to PclService you need to add an extra to the Intent used to start or bind to the service:

```
private void startPclService() {
  if (!mServiceStarted)
  {
    String logDirectoryPath =
```

```
   Environment.getExternalStoragePublicDirectory(
   Environment.DIRECTORY_DOWNLOADS
   ).getAbsolutePath(); // Download Directory Path
    Intent i = new Intent(this, PclService.class);
    i.putExtra("LOGGER_HOME_DIR", logDirectoryPath); // change logs
location

    if (getApplicationContext().startService(i) != null)
      mServiceStarted = true;
  }
}
```

The default location is the same than previous Add-on versions and don't need permission to add.

3_7 Using PclService

Once your application bound to the PclService, you will get a PclService interface and you will be able to use the PCL services.
The example below shows how to use PclService to print a text on the Companion printer:

```
public boolean printText( String strText ) {
  boolean Result = false;

  byte[] PrintResult = new byte[1];

  Result = mPclService.printText(strText, PrintResult);

  return Result;
}
```

As services provided by PclService interface are long-running services (because they mostly use Bluetooth communication), you must call these services in a thread outside the UI thread to avoid ANR (Application Not Responding) issues.
To achieve this you can either use standard Thread or AsyncTask for instance, as described in the code below:

```
class DoTransactionTask extends AsyncTask<Void, Void, Boolean> {
  // declare the parameters that will be used as input and output
  private TransactionIn transIn;
  private TransactionOut transOut;
  public DoTransactionTask(TransactionIn transIn, TransactionOut transOut)
{
    // initialization in the class constructor
    this.transIn = transIn;
    this.transOut = transOut;
  }

  protected Boolean doInBackground(Void... tmp) {
    // run the actual PCL function
    Boolean bRet = doTransaction(transIn, transOut);
    return bRet;
  }
```

```
  protected void onPostExecute(Boolean result) {
    // do the analysis of the returned data of the function
  }
}
```

For more details about AsyncTask please read Android developer documentation here.
Methods defined by the IPCLService interface are described in the javadoc included in “Add-on PCL for Android”.

3_8 Checking connection state

An Android application can check the TCP connection state between the Android device and the Telium Companion in either ways:

- Call the PclService serverStatus method on a regular basis
- Implement a broadcast receiver that will listen for `"com.ingenico.pclservice.intent.action.STATE_CHANGED"` and get the extra string `"state"` when the broadcast is received. The string value can be `"CONNECTED"` or `"DISCONNECTED"`.

```
public class YourActivity extends Activity {

  // Declare broadcast receiver
  private StateReceiver m_StateReceiver = null;

  // Implement broadcast receiver
  private class StateReceiver extends BroadcastReceiver
  {
    @SuppressLint("UseValueOf")
    public void onReceive(Context context, Intent intent)
    {
      String state = intent.getStringExtra("state");
   // Do appropriate action depending on state
    }
  }

  // Initialize broadcast receiver
  // This can be called in onResume for instance
  private void initStateReceiver()
  {
    if(m_StateReceiver == null)
    {
      m_StateReceiver = new StateReceiver(this);
      IntentFilter intentfilter = new
IntentFilter("com.ingenico.pclservice.intent.action.STATE_CHANGED");
        registerReceiver(m_StateReceiver, intentfilter);
    }
  }

  // Deinitialize broadcast receiver
  // This can be called in onPause for instance
  private void releaseStateReceiver()
```

```
  {
    if(m_StateReceiver != null)
    {
      unregisterReceiver(m_StateReceiver);
      m_StateReceiver = null;
    }
  }
}
```

4 Developing your application

This section will focus only on how to use the various features provided by the “Add-on PCL for Android” to successfully use the features of the payment terminal.
It will give a brief overview of the functions to use.

4_1 Doing a payment

This section does not apply to Axium Terminals.

We can define 3 main actions for the payment transaction:
- Trigger a payment transaction (for a specific amount and using a specified currency) on the payment terminal
- Get the signature of the customer on the mobile device screen and add it to the payment transaction parameter, if a signature is requested by law or depending on the type of card used
- Print the transaction receipt. The print of the receipt can be done either by the payment application or by the application on the mobile computer. If the print of the receipt is not done by the payment application, or if you want to print another type of receipt, for example, a bill or a loyalty coupon, you can do it from the application on the mobile computer

4_1_1 Sending the payment request

Depending on the kind of payment application that is running on the payment terminal, you may want to use one of the 2 functions:
- doTransaction(TransactionIn, TransactionOut)
- doTransactionEx(TransactionIn, TransactionOut, appNumber, extDataIn, extDataInSize, extDataOut, extDataOutSize);

Where:
- TransactionIn is a TransactionIn object that contains the parameters of the payment transaction to be executed
- TransactionOut is a TransactionOut object that contains the result of the payment transaction that has been executed

The first one is the simplest one. You just have to fill the amount and the currency of the transaction and the rest will be automatically managed on the payment terminal side.
However, in some cases, additional data are requested to be able to process the transaction. In this case, the apiDoTransactionEx should be used, as it provides a space to put additional data and to select a specific payment application.

The possible payment transactions that are supported by these functions are:
- Debit
- Credit
- Annulation of a previous transaction
- Duplicata of a previous transaction

In the extended version of the function, the meanings of the additional parameters are:
- appNumber: application number to be executed on Telium – allow to force the use of a specific payment application

- extDataIn and extDataInSize: input buffer for sending a block of data to the payment application
- extDataOut and extDataOutSize: output buffer for receiving a block of data from the payment application

TransactionIn and TransactionOut are parcelable objects defined in TransactionIn.aidl and TransactionOut.aidl and implemented in Transaction.jar.

4_1_2 Capturing the signature

If a signature is needed for the payment to complete, the payment application on Ingenico terminal side has to call the specific function PDA_do_signature_capture, PDA_getSignature or PDA_catchSignature. The latter function doesn't require the signature to be sent to the companion.
If you develop your application with PclService.apk, the PCL layer will by default automatically manage requesting the signature on mobile device side, capturing it and sending the data back to the payment application as a bitmap image.
If you want the application to manage the signature capture you will need to call PclService registerCallback method with externalSignCap parameter set to true.
If you develop your application with PclServiceLib.jar, the application must deal with signature capture. It must previously call PclService registerCallbak method.

4_1_3 Printing the receipt

In order to print a receipt on the Companion printer (either integrated printer on iWL or Bluetooth printer on iSMP or iCMP Companion), you need to follow these steps:
- Open the connection between the mobile device and the printer using the openPrinter function
- Send the information to be printed (using one or more print commands)
- Close the communication channel with the printer, using the closePrinter function. This last step is very important and it is mandatory to do it as soon as you are done with your printing

The various possible print commands are the following:
- printText to print one or several lines of text
- printBitmap to print a bmp file (passed as a byte array)
- printBitmapObject to print an android.graphics.Bitmap object
- printLogo to print a logo

You can also set Telium printer font with setPrinterFont API.

A logo is a bitmap file that is stored in the Telium terminal. You can either store this logo permanently as a file with extension .LGO in /HOST directory of the terminal or using the storeLogo function. In the latter case the logo is stored in volatile memory so you'll have to store it again when the Companion restarts.
If you know that you will always print the same bitmap on your receipt, then you can use the storeLogo function to send the data once on the terminal and later, just use the printLogo function. Doing so, you will not have to send the bitmap data from the mobile device to the payment terminal for each receipt or bill you want to print.

In order to print accentuated characters, you have to embed standard font ISO15 in the terminal (this is the default font used on terminal side).
Alternatively you can embed other ISO-8859 fonts and use setPrinterFont API to select the appropriate one.

4_1_4 Exchanging additional information with the payment application

Three functions are provided, allowing you to exchange information (either or not payment related) between the application running on the mobile device and the application running on the payment terminal.
Use sendMessage function to send data to the payment terminal and receiveMessager function to receive data from the terminal.
The receiveMessage function is a non-blocking function.
The flushMessages function is used to erase the unread messages sent by the terminal.

4_1_5 Generating a receipt on smartphone side (1.04 and above)

Starting with add-on 1.04 and above, it is possible, to generate the receipt on Android side, if the Companion that you are using is not connected to a printer (either physically like in the case of an iWL or using a Bluetooth connection).
Generated the receipt on Android can be needed in many use cases:

- The receipt needs to be sent via email to the customer
- The printer is linked to the smartphone so the receipt must be generated on the smartphone
- The receipt information must be sent to a transaction server

If you want to manage the receipt from the smartphone application, you must do the following:

1. Implement callback functions provided by PclService
2. Register these callback functions to use them
3. Unregister callback functions when they are not needed

The following callback functions should be implemented to properly generate the receipt:

- shouldFeedPaper: tells the application that one or more blank lines must be generated
- shouldPrintText: tells the application that some text must be printed
- shouldPrintImage: tells the application that an image must be printed
- shouldStartReceipt: tells the application that a receipt starts
- shouldEndReceipt: tells the application that a receipt ends
- shouldCutPaper: tells the application that printing session ends
- shouldPrintRawText: tells the application that the text must be printed with a character set defined by the Telium application.

To keep compatibility with existing applications both callbacks shouldPrintRawText and shouldPrintText are called (in this order) when a Telium application asks for text printing. This allows Android applications handling properly cases where unsupported character set is used in the Telium application. An example is provided in PclTestApp application.

For more details about these functions refer to PclService javadoc.

These 3 steps have to be implemented.

The example below shows how to implement this:

```
private IPclServiceCallback mCallback = new IPclServiceCallback() {
  public void shouldFeedPaper(int lines) {
    // Update application TextView for instance
  }

  //Implement all other callbacks
```

```
}

// Somewhere in your application you need to register the callback
// functions to be able to use them
mPclService.registerCallback(mCallback);

// Somewhere in your application you need to unregister the callback
// functions when you don't need them
mPclService.unregisterCallback(mCallback);
```

4_2 Managing the terminal

This section does not apply to Axium Terminals.

The main terminal management functions that can be used are the following:

- Get or set the date / time of the payment terminal using getTime and setTime
- Launch an upgrade, using the doUpdate function. This will trigger the update function on the terminal, meaning it will automatically connect to the Terminal Management System (if one defined) and download the available upgrades (if any)
- Reset the terminal, using the resetTerminal function
- Get the status of the printer, using the getPrinterStatus
- Get terminal components, especially the SPMCI version
- Control backlight

4_3 Using the IP link between the Android and Telium

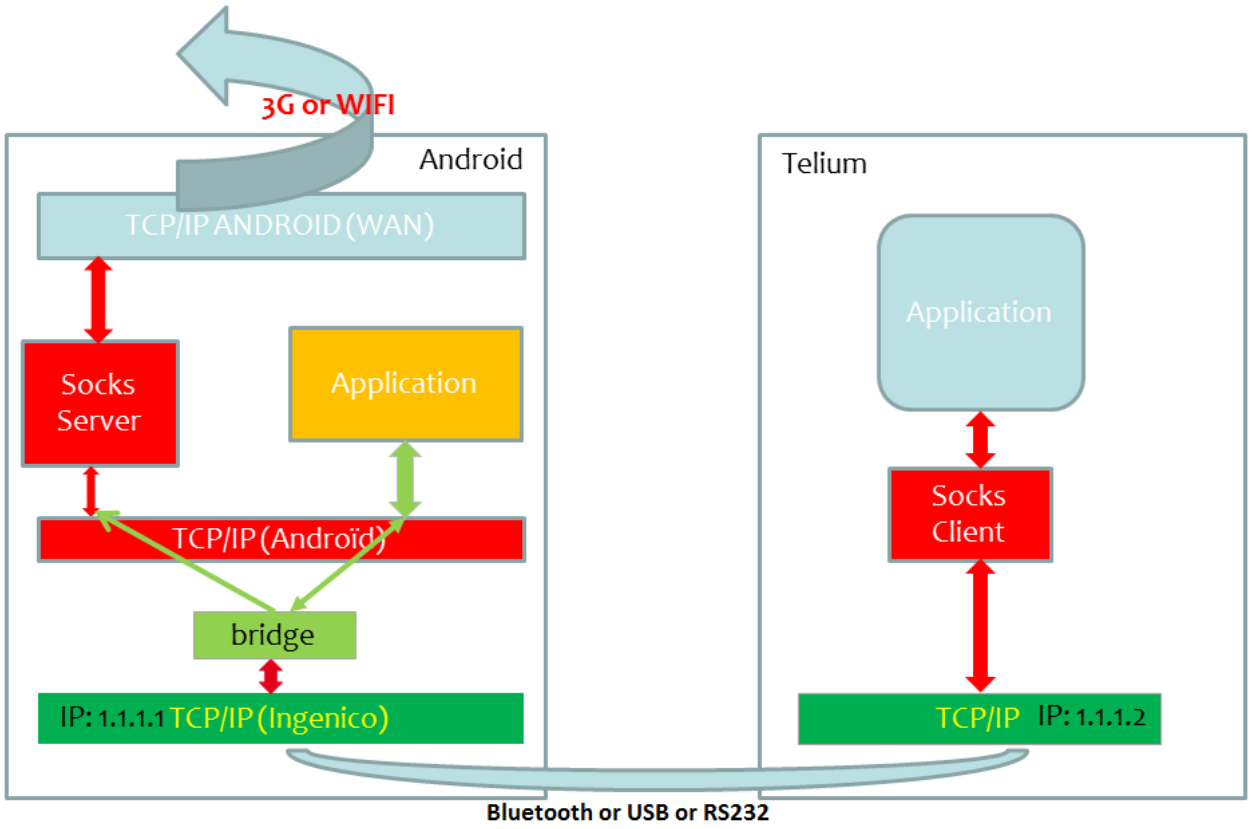
One of the features provided by the "Add-on PCL for Android" is the creation of an IP link between the Telium and the Android devices.
This can be useful if you want to reuse an existing Telium application that acts as an IP server. In this case, the application can be reused as-is, without any modifications.
The most common example is the C3 connectivity. But any other IP-compatible application (RAM for example) can be reused.

This section does not apply when the Terminal is connected using Wi-Fi or Ethernet.

The created link is a standard IP layer and all SSL and TCP standard features can be used, through the standard Ethernet channel provided by the LinkLayer. This link can be used in both directions:

- connect from Telium side to a server located on Android
- connect from Android side to a server located on Telium

The architecture of the IP link is described in the figure below:



As shown in the diagram, the connection is technically not direct between Android and Telium: it must go through a bridge first that will detect whether:

- The connection must be forwarded to Telium (if coming from Android)
- The connection must be forwarded to Android (if coming from Telium)
- The connection must be forwarded to the WAN (if coming from Telium)

The bridge must be configured before an Android application can establish a TCP connection with the Telium application.

Note that some TCP ports are reserved for PCL service and must not be used by another Android application: 5184, 5186, 5188 and 1080.

4_3_1 Configuring the bridge

To add a new port to the bridge, add the following code in your Android application (for example in the piece of code where the PCLService is bound, to have it executed only once at the beginning of the application):

```
public void onServiceConnected(ComponentName className,
        IBinder boundService )
{
  mPclService = IPclService.Stub.asInterface((IBinder)boundService);
  if( mPclService != null ) {
  try {
    mPclService.addDynamicBridge(9599, 0);
  } catch( RemoteException ex ) {
```

```
    ex.printStackTrace();
  }
}
```

The above code declares that all requests to the port 9599 will be forwarded by the bridge to the Telium application (which in turn is supposed to be listening on this port number).
The first parameter of addDynamicBridge function is the TCP port and the second parameter is the direction of the connection (0 means from Android to Telium and 1 means from Telium to Android).

Starting with add-on 1.05.01, you can also use addDynamicBridgeLocal method to allow only the localhost connections to be forwarded to the Telium application.

4_3_2 Connecting from Android to Terminal

In order to successfully call a TCP server located on Telium, the following actions must be performed:
- Have the TCP server listening on port [XXXX] on Telium
- Add the [XXXX] port value to the bridge configuration with addDynamicBridge function (with second parameter set to 0)
- On Android, open the connection using the Android localhost IP address (127.0.0.1) and the [XXXX] port number

4_3_3 Connecting from Terminal to Android

In order to successfully call a TCP server located on Android, the following actions must be performed:
- Have the TCP server listening on port [XXXX] on Android
- Add the [XXXX] port value to the bridge configuration with addDynamicBridge function (with second parameter set to 1)
- On Telium, open the connection using the Android device IP address (1.1.1.1) and the [XXXX] port number

4_3_4 Connecting from Terminal to remote host through Android socks server

To be able to use the communication features provided by the Android device, you need to configure the Telium Manager to use a Proxy.
To configure the proxy, follow the steps:
- Enter the menu in Telium manager: F > Telium Manager > Initialization >Hardware > Proxy Setup
- Enter the following parameters:
  - USE PROXY ?: 2 - Yes
  - USE PROXY ?: SOCKS5
  - IP ADDRESS – PROXY: 1.1.1.1
  - PORT - PROXY: 1080
  - USE AUTHENTICATION: No

Proxy applies only to Telium 2.

Your Terminal application can use standard TCP/IP functions to connect to remote host (either BSD or LinkLayer functions).
Note that this architecture is also compliant with the use of SSL connections. Since the Android device is only providing the network connectivity, certificate management must be done on Telium side.

Starting with add-on 1.05, some new features are available:

- It is possible to connect to 5 remote hosts simultaneously (instead of 1 in previous versions). This especially allows doing FTP connections (in active mode only) from Telium to remote FTP servers.
- Android socks server implements name resolution so it is possible to use hostname instead of dotted IP address (only when using LinkLayer)

Starting with add-on 1.06.01 and Telium SDK 9.22, it is possible to make IP lookup and reverse lookup from a Terminal application with respectively DNS_GetIpAddress and DNS_GetDomainName functions.

In your Telium application, you cannot check for the availability of the Ethernet link using the standard IsETHERNET function. In order to know whether the IP link is available, you must use the EXT_Available function.

4_3_5 SSL considerations

One important point to note is that the when PCL is used, the fact that there is a component between the Companion and the Internet (or LAN), is absolutely transparent. It as
If an SSL connection needs to be established with a remote server, the certificates used will be those of:

- The Telium Companion
- The remote server

The Android device does not need to store or use any certificates, as it is not the one in charge of encrypting or decrypting the communication. This also means data being transferred over the SSL channel cannot – by any means, including RAM scrapping - be read at the smartphone level because they are just passing through, encrypted.

Please note that SSL v3 is now deprecated due to “Poodle” issue (10/2014).
Mobile Payment Application Developers must update and distribute a new version of applications with a new protocol mask that allows TLSv1.2 and TLSv1.1 and forbids TLSv1.0 and all SSL versions.
For example, Telium Payment Applications using PCL Communication channel to connect their payment gateway must now comply with this requirement.

4_4 Using the barcode reader

This section applies only to Telium 2.

When your Android device is connected to an iMP352 Companion you can use the barcode reader of the Companion for reading barcodes.
To allow barcode reader usage you have to:

- Open barcode reader peripheral using the openBarcode or openBarcodeWithInactivityTo method
- Declare a broadcast receiver to receive barcode events asynchronously
- Optionally configure barcode reader with the following functions:
  - bcrSetReaderMode
  - bcrSetGoodScanBeep
  - bcrEnableTrigger
  - bcrSetImagerMode
  - bcrSetBeep
  - bcrSetLightingMode
  - bcrSoftReset
  - bcrEnableSymbologies
  - bcrDisableSymbologies
  - bcrSetNonVolatileMode
  - bcrSetSettingsVersion
  - bcrGetSettingsVersion

You can also use utility functions bcrGetFirmwareVersion, bcrSymbologyToText, bcrStartScan and bcrStopScan.

The broadcast receiver will listen for:

- "com.ingenico.pclservice.action.BARCODE_EVENT" and get the extra byte array "barcode" and the extra integer "barcode_symbology" when data is received.
- "com.ingenico.pclservice.action.BARCODE_CLOSED" when barcode reader is closed on inactivity timeout

The example below shows how to implement a broadcast receiver to receive barcode events:

```
public class YourActivity extends Activity {

  // Declare broadcast receiver
  private BarCodeReceiver m_BarCodeReceiver = null;

  // Implement broadcast receiver
  private class BarCodeReceiver extends BroadcastReceiver
  {
    @SuppressLint("UseValueOf")
    public void onReceive(Context context, Intent intent)
    {
      String action = intent.getAction();
      if (action.equals("com.ingenico.pclservice.action.BARCODE_CLOSED"))
      {
        ViewOwner.onBarCodeClosed();
      }
      else
```

```
      {
        byte abyte0[] = intent.getByteArrayExtra("barcode");
        String BarCodeStr = new String(abyte0);
        int symbology = intent.getIntExtra("barcode_symbology", -2);

        ViewOwner.onBarCodeReceived(BarCodeStr, symbology);
      }
    }
  }

  // Initialize broadcast receiver
  // This can be called in onResume for instance
  private void initBarCodeReceiver()
  {
    if(m_BarCodeReceiver == null)
    {
      m_BarCodeReceiver = new BarCodeReceiver(this);
      IntentFilter intentfilter = new
IntentFilter("com.ingenico.pclservice.action.BARCODE_EVENT");

intentfilter.addAction("com.ingenico.pclservice.action.BARCODE_CLOSED");
   registerReceiver(m_BarCodeReceiver, intentfilter);
    }
  }

  // Release broadcast receiver
  // This can be called in onPause
  private void releaseBarCodeReceiver()
  {
    if(m_BarCodeReceiver != null)
    {
      unregisterReceiver(m_BarCodeReceiver);
      m_BarCodeReceiver = null;
    }
  }
}
```

You can find a complete example of barcode reader usage in PclTestApp application source code.

When using openBarcode function to open the barcode reader, a default 10 minutes inactivity timeout is applied. You can also define your own inactivity timeout with openBarcodeWithInactivityTimeout function. If the barcode is not used during the defined timeout, it will be automatically closed and an event will be sent.
If you set inactivity timeout to 0, it means that the barcode reader won't be closed automatically on inactivity timeout.
To avoid useless power consumption, make sure in your application to close the barcode reader when it is no longer needed and to reopen it only when necessary.

The barcode reader can be configured to read all or a subset of all available barcode types. To enable all barcode types, the "**ICBarCode_AllSymbologies**" constant can be used. This, however, may have a severe impact on performance, since the more barcode types are enabled, the lower the decoding performance will be. So, **a good practice would be to enable only the subset of barcode types that have to be read**.

Power Management Considerations :

The barcode reader is the most consuming component of the Companion, and as such, Android applications should use it very carefully and make sure to optimize power consumption to save battery. To achieve this, few guidelines have to be followed:

- **Turn on the barcode reader only when it is required** to scan barcodes and turn it off as soon as it is no more needed. The barcode reader has high power consumption, even when not used for barcode reader.
- Use single scan mode whenever possible. This mode turns off the light of the barcode reader when a barcode is read
- **Use the 1D imager mode whenever possible**. This mode uses a small light beam.
- The illumination level can also be reduced using the barcode readers API, provided that this doesn't decrease the performance.

4_5 Remote download

The remote download feature relies on Telium existing features, so nothing specific has to be done from an integration perspective. You just need to ensure that connectivity parameters are properly set:

- Proxy (see section 4_3_4)
- TMS network: F > Initialization > Parameters > T.M.S.
  - In the TMS Access menu, the “IP / Eth” option must be selected

It is now possible to read and write TMS parameters from the mobile device with respectively tmsReadParam and tmsWriteParam functions.

When the “update” command is sent by the mobile device, the PCL part that is located in Telium will automatically call the callHost function that, in turn, will connect to the TMS server and update the terminal.

4_6 Implementing the Local Download feature

The “Add-on PCL for Android” is compatible with the use of the TMS ECR Agent.
TMS ECR Agent is a tool used to update Ingenico terminals using files that are stored locally on the mobile device (unlike the remote download, where Ingenico terminal remotely connects to the TMS sever to get updates).
This feature is delivered as a separate Android package and can be included in your Android application.
Contact your Ingenico representative to get the TMS ECR Agent.

The TMS-Agent is provided as a native dynamic library. This library can be used from an Android application thanks to its JNI interface.

4_6_1 Integration

The integration of TMS-Agent is done in two steps:

Step 1:
A “`TmsAgent`” class needs to be implemented in the Java application to interface the JNI function. Here is the full class implementation:

TmsAgent.java

```
package com.ingenico;

public class TmsAgent {
   /** Native method */
   public native int tms(String param);

   /** Load JNI .so on initialization */
   static { System.loadLibrary("tms_android"); }
}
```

It is important to respect the package name (com.ingenico), the class name (TmsAgent) and the function name (tms).

Step 2:
Copy the “`tms_android.so`” native library in the “`libs/`*`xxx`*” folder of the Android application, where “*`xxx`*” is the platform you are targeting (e.g. “`armeabi`”).

4_6_2 Usage

Parameter: A formatted String with the same options as for the executable integration. See TMS ECR Agent document for more information.

Return: 0 in case of successful processing.

Download may take time. It is a best practice to not call the TMS-Agent from the main UI thread.

Sample code

```
/* Start a remote download */
TmsAgent tmsagent = new TmsAgent();
int ret = tmsagent.tms("-no_retail_proto -tms ip:10.1.2.3:1234 -com
ip:localhost:5678");
```

4_6_3 Testing the LocalDownload feature

PCL sample application PCLTestApp provides a demonstration of the LocalDownload feature. However, to make it work you need to do the following

On Telium
1/ Upload the needed files for local download management
Use LLT to copy the following from the TMS ECR Agent package

- Telium > Local > LocalDownload > Sign > 8442340xxxx.AGN file (use the catalog that matches your Telium device)
- Telium > Local > LD_Test > LD_TEST.AGN file (use the catalog that matches your Telium device)

Using LLT, copy into the HOST directory of the Telium device the file:

- Telium > Local > Config > LOCALDOWNLO.PAR

2/ Make the Telium terminal ready to accept incoming files
From the manage menu, go to LD_TEST, then

- Select **Switch to ETH**
- Select **Start LD**

Creation of the file that will be downloaded
Use the IngPackager tool to create the zip file that contains the update to send to Telium. Make sure to select the Telium architecture for the package!

On Android

- Create a temp directory at the root of the tablet tree directory
- Copy the zip file that contains the file to be uploaded to Telium
- Start the PCLTestApp application
- Go to Unitary Tests > Update > U001 Local Update TMS Agent
- In the “File to download” field, enter the name of the zip file that you copied into the temp directory. Don’t forget to add the .zip extension at the end
- Click on the Run button: on the Telium side, you will see the progress of the download

4_7 Internationalization

When you create an Android application, a file ./res/values/strings.xml is automatically created. This file should be used to store all the text strings that are displayed by the application.
If you want your application to support another language (i.e. to display the text in another language), you just have to perform the following steps:

- In the application source folder, go to the *res* directory and create a sub-directory named *values-[CC]* where [CC] is the code of the language you want to support, as defined in the ISO-639-1 code list.
- Copy the existing strings.xml files from the *res/values* directory into the new directory
- Translate it into the needed language which code is [CC]

When the Android device language will be changed, the strings will be automatically taken from the correct resource files.

4_8 Signing the application

All Android applications must be digitally signed to be installed. By default, the Android SDK contains a test certificate that is used during the built of the application. This certificate is specific to an SDK environment (i.e., the developer's computer).
Building the same application on another computer will result in a different signature and at installation time, Android OS will detect this application as an entirely new application.
To avoid this, it is strongly recommended to sign the application with a real certificate, not the one provided by default with Android SDK.
The certificate can be a self-signed certificate. Just make sure to follow the recommendation from Android website: http://developer.android.com/tools/publishing/app-signing.html

5 PCL File Sharing Features

This section does not apply to Axium Terminals.

5_1 Introduction

One of the new features of the Add-on PCL for Android 1.14.00 is allow file sharing between an Android device and a Terminal device. The “Add-on PCL for Android” contains a sample developed in Kotlin language to give you an example of an implementation of the PCL File Sharing features.

This features works only with the Tetra terminal.

5_2 Shared Instance

The PCL File Sharing class is developed around a unique shared instance which will allows you to get this instance from anywhere in your code.
You can get it by calling static method *getSharedInstance()*.

5_3 Start File Sharing

The PCL File Sharing feature embed a server which allows the transfer of files between Android and Terminal. Three method exist to start the file sharing server.

5_3_1 Simple start

You are able to only start the server by calling:

Sample code in kotlin

```
PCLFileSharing.getSharedInstance().start(8000, true) { result ->
            if (result == PCLFileSharingResult.PCLFileSharingResultOk) {
                // Do what you want here
            }
}
```

This method is asynchronous and will be execute on the default priority. Once the server is launched, the completion handler will be called, and you will be able to execute your code.
You will need to perform a Remote Upgrade from the terminal in order to connect it to your Android application.

5_3_2 Start and DoUpdate

This method allows you to setup the terminal with the current information of your Android application and automatically start the Remote Upgrade of the terminal in order to connect it to your Android application.

Sample code in kotlin

```
PCLFileSharing.getSharedInstance().startAndLaunchDoUpdate(this,"pairingfile.txt",
8000) { result ->
```

```
            if (result == PCLFileSharingResult.PCLFileSharingResultOk) {
                // Do what you want here
            }
}
```

This method will use the PclService methods in order to get the TMS information of the terminal, save them, set the TMS information of the terminal with the information provided to the method.

Then, the method will start the server and perform a Do Update command in order to start the Remote Upgrade of the terminal.

This method is asynchronous and will be execute on the default priority. Once the server is launched, the completion handler will be called, and you will be able to execute your code.

Before starting FileSharing, you need to verify on terminal that TMS configuration “NetworkType” is correctly configured for the communication protocol that the Terminal is using (Control Panel -> Software management -> Evolution -> Remote upgrade -> Configuration).

Terminal connectivity	**NetworkType**
Bluetooth/USB/RS232 through PCL	PCL
Wi-Fi	WiFi
Ethernet	ETHERNET

5_3_3 Start in LLT mode

You are able to start server with a Terminal in LLT mode. This method will only works on a USB communication and on the LLT mode of the connected Terminal.

Sample code in kotlin

```
PCLFileSharing.getSharedInstance().startAsLLTMode(this,"pairingfile.txt")
{ result ->
            if (result == PCLFileSharingResult.PCLFileSharingResultOk) {
                // Do what you want here
            }
}
```

This method is asynchronous and will be execute on the default priority. Once the server is launched, the completion handler will be called, and you will be able to execute your code.

5_4 List Terminal File

You will be able to get the files in the directories of the terminal in order to create your user interface for example

Sample code in kotlin

```
PCLFileSharing.getSharedInstance().list(“/”) { files ->
     // Do what you want here
}
```

This method is asynchronous and will be execute on the default priority. Once the server is launched, the completion handler will be called, and you will be able to execute your code.

5_5 Upload file

You can send a file from your Android application to the Terminal device by calling:

Sample code in kotlin

```
PCLFileSharing.getSharedInstance().upload("SendFile.txt", "/import"){result ->
          if (result == PCLFileSharingResult.PCLFileSharingResultOk) {
              // Do what you want here
          }
}
```

This method is asynchronous and will be execute on the default priority. Once the server is launched, the completion handler will be called, and you will be able to execute your code.

5_6 Download file

You can download a file from Terminal to your Android application by calling:

Sample code in kotlin

```
PCLFileSharing.getSharedInstance().download("/path/to/the/file/to/download",
"/path/where/save/file") { result ->
          if (result == PCLFileSharingResult.PCLFileSharingResultOk) {
              // Do what you want here
          }
}
```

This method is asynchronous and will be execute on the default priority. Once the server is launched, the completion handler will be called, and you will be able to execute your code.

5_7 Stop File Sharing

At the end you need stop the server by calling:

Sample code in kotlin

```
PCLFileSharing.getSharedInstance().stop() { result ->
             if (result == PCLFileSharingResult.PCLFileSharingResultOk) {
                 // Do what you want here
             }
}
```

This method is asynchronous and will be execute on the default priority. Once the server is launched, the completion handler will be called, and you will be able to execute your code.
This method valid packages for installation by default. With option,

When you stop the server, the terminal can reboot if need.
If you start server with method *`startAndLaunchDoUpdate()`*, when you stop it, the TMS parameters are restore automatically.

6 Appendix: Sample applications in Android Studio

Sample applications provided in this add-on have been developed on Android Studio.

Use Android Archive Library (aar) in Android Studio

If you use the Android Archive Library (aar) with sample or your application, you need to add this following packaging options:

```
packagingOptions {
        pickFirst "lib/*/libtlvtree.so"
        pickFirst "lib/*/libpcltools.so"
}
```