

# Add on PCL for iOS

## Integration Guide

**ICO-OPE-02151 V2**  
**Restricted**

**Ingenico**  
2014/11/24

[www.ingenico.com](http://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

<b>1 Introduction.....</b>	<b>5</b>
1_1 Acronyms and terminology.....	5
<b>2 Installation.....</b>	<b>6</b>
2_1 Add-on content.....	6
2_2 Prerequisites .....	6
2_2_1 Hardware prerequisites .....	6
2_2_2 Software prerequisites.....	6
2_2_3 Supported versions of iOS .....	6
<b>3 Setting up the development environment.....</b>	<b>8</b>
3_1 Structure .....	8
3_2 Installation.....	8
3_3 Dependencies .....	9
3_4 Naming Convention.....	10
3_5 Application Requirements .....	10
3_6 Using the 64-bit version of the PCL framework.....	11
<b>4 Developing your application .....</b>	<b>13</b>
4_1 Architecture.....	13
4_2 General principles .....	13
4_2_1 Delegate .....	13
4_2_2 Additional Compiler Flags When Using Categories .....	14
4_2_3 Managing background state .....	14
4_3 Checking the connection state between the device and the Companion .....	14
4_4 Selecting the Telium device to use .....	16
4_5 Doing a payment.....	17
4_5_1 Sending the payment request .....	17
4_5_2 Capturing the signature .....	20
4_5_3 Printing the receipt.....	20
4_5_3_1 Printing Text .....	21
4_5_3_2 Printing Images.....	23

4_5_3_3 Feeding Paper .....	23
4_5_3_4 End of Printing .....	23
4_5_4 Exchanging additional information with the payment application .....	23
<b>4_6 Managing the terminal.....</b>	<b>24</b>
4_6_1 Terminal information .....	24
4_6_2 Terminal management .....	24
4_6_3 Remote download.....	25
4_6_4 Power Management .....	26
4_6_4_1 Backlight & Suspend Timeouts .....	26
4_6_4_2 Companion Battery Level .....	26
<b>4_7 Using the network connection of the iOS device .....</b>	<b>27</b>
4_7_1 Prerequisites .....	28
4_7_2 Connecting from iOS to Telium .....	29
4_7_3 Connecting from Telium to iOS .....	29
4_7_4 Connecting from Telium to remote host through iOS socks server .....	29
4_7_5 Remote host connection state .....	30
4_7_6 ICPPPDelegate Protocol .....	30
<b>4_8 Using a generic COM port.....</b>	<b>31</b>
4_8_1 Synchronous Approach .....	31
4_8_2 Asynchronous Approach .....	31
4_8_3 ICTransaction Initialization .....	31
4_8_4 ICTransaction Disposal .....	32
4_8_5 State Changes .....	32
4_8_6 Sending/Receiving Data Using RAW Streams .....	32
4_8_6_1 Initialize Read and Write Buffers .....	32
4_8_6_2 Writing Data .....	32
4_8_6_3 Reading Data .....	33
<b>4_9 Using the barcode reader.....</b>	<b>34</b>
4_9_1 Initializing the barcode reader .....	34
4_9_2 Barcode reader configuration .....	34
4_9_2_1 Barcode reader default configuration .....	36
4_9_2_2 Barcode reader methods .....	36
4_9_2_3 Barcode reader retrieve configuration methods .....	37
4_9_3 Reading barcodes .....	37
4_9_4 Releasing the barcode reader .....	38
4_9_5 Additional functions .....	38
4_9_6 Power Management Considerations .....	39
<b>4_10 Using the Virtual SPP Channel .....</b>	<b>39</b>
4_10_1 ICSPSP Initialization .....	39
4_10_2 ICSPSP Disposal .....	40
4_10_3 State Change .....	40
4_10_4 Exchanging Data .....	40
<b>4_11 Using an external Bluetooth printer.....</b>	<b>40</b>
4_11_1 Naming Conventions .....	40
4_11_2 Application Requirements .....	40
4_11_3 Bluetooth Printing API .....	41
4_11_4 ICAdministration Initialization .....	41
4_11_5 Constants & Status Codes .....	41
4_11_6 API Functions .....	41

4_11_6_1 Open Printer .....	42
4_11_6_2 Close Printer .....	42
4_11_6_3 Print Text .....	43
4_11_6_4 Print Bitmap .....	43
4_11_6_5 Print Bitmap, Size, Alignment .....	44
4_11_6_6 Store Logo .....	44
4_11_6_7 Print Logo .....	45
 4_12 Debugging your application .....	 45
 4_13 Sample code .....	 47
 4_14 Using the iOS simulator .....	 47
 <b>5 Signing the application .....</b>	 <b>48</b>
 <b>6 FAQ .....</b>	 <b>49</b>
 6_1 File Too Small when Compiling an App .....	 49
 6_2 Application not Compiling Properly – Missing Symbols .....	 49
 6_3 ICAdministration Methods Unrecognized at Runtime .....	 50
 6_4 Wrapping iSMP Library Objects with Singletons .....	 51
 6_5 Double Definition for Class ICDevice .....	 51

# 1 Introduction

This document will help you develop an iOS-based applications using “Add-on PCL for iOS”.

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

This document is compatible with the PCL add-on v 2.10.00.

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

<b>PCL</b>	Payment Communication Layer
<b>Companion (or Telium device)</b>	Ingenico device that is used to manage the payment. It can be either the secure part of an iSMP, an iSMP Companion, or an iCMP
<b>Telium</b>	Companion internal operating system
<b>iOS device</b>	Any device running under iOS system: iPhone, iPod Touch or iPad



This symbol indicates an important Warning.



This symbol indicates a piece of advice.

## 2 Installation

### 2\_1 Add-on content

The “Add-on PCL for iOS” contains all the binaries needed to ensure the communication between an iOS device and a Companion, in order to perform payment transactions or Companion administration. Before proceeding with the installation steps, you must extract the content of the Add-on on a Windows computer in a %Add-on\_Dir%.

### 2\_2 Prerequisites

#### 2\_2\_1 Hardware prerequisites

The “Add-on PCL for iOS” can be used with the following Telium device (Companion):

- iMP350 (iSMP with barcode reader)
- iMP320(iSMP without barcode reader)
- iMP322 (iSMP Companion without barcode reader)
- iMP352 (iSMP Companion with barcode reader)
- iCM122 (iCMP)

#### 2\_2\_2 Software prerequisites

SDK installed on the Telium device must be:

- 9.20 or higher

The following SDK components must be downloaded on the Telium device:

- OS
- Manager
- Link Layer With IP (component has been merged in the Manager since SDK 9.16)
- SPMCI

It is possible but not recommended to use older SDK version (up to 9.12) if you don't need the latest PCL features.

Note that some features from earlier SDK will be deprecated so it is strongly advises to use at least SDK 9.18 with this version of PCL add-on.

If you want to use the backward compatibility provided by this add-on with previously developed Telium applications, you need to add the following component:

- DLL TCP/IMP



---

**A computer running under Microsoft Windows® operating system is mandatory for the installation of the system: the “Add-on PCL for iOS” is delivered as a Windows installation file and the LLT tool (to access Companion memory) is only Windows compliant**

---

#### 2\_2\_3 Supported versions of iOS

iOS devices must run under OS version 5.1.1 or higher.



iOS 7.1 and 7.1.1 should be used cautiously due to an issue related to reading/writing data during exchanges between an iOS device and a Telium one.

---

## 3 Setting up the development environment

PCL is a set of libraries that must be used to develop the sale application.

The PCL API is packaged as an iOS framework that can be easily integrated into Xcode. iOS framework is the conventional way to distribute static libraries in iOS/OS X environments. They consist of an ordinary folder with the **“.framework”** extension that contains all the library's files (binary and headers).

To be able to use the PCL features in your application, copy in your development environment the iSMP.framework from the %Add\_onDir%/iOS directory.

### 3\_1 Structure

The **iSMP.framework** package contains the following:

- **“Headers”** folder containing the library's headers files,
- The binary file of the library that is name like **“libiSMP-x.y.a”** where **“x.y”** represents the release version,
- A symbolic link **“iSMP”** that has the same name as the framework. This link points to the library's binary file and is used in order to avoid naming the binary after the framework's name.



Sometimes, when moving the framework package from Windows to Mac, the **symbolic link is broken** and needs, therefore, to be re-created in order to compile properly. Please, refer to the “FAQ” section to see how to solve this issue.



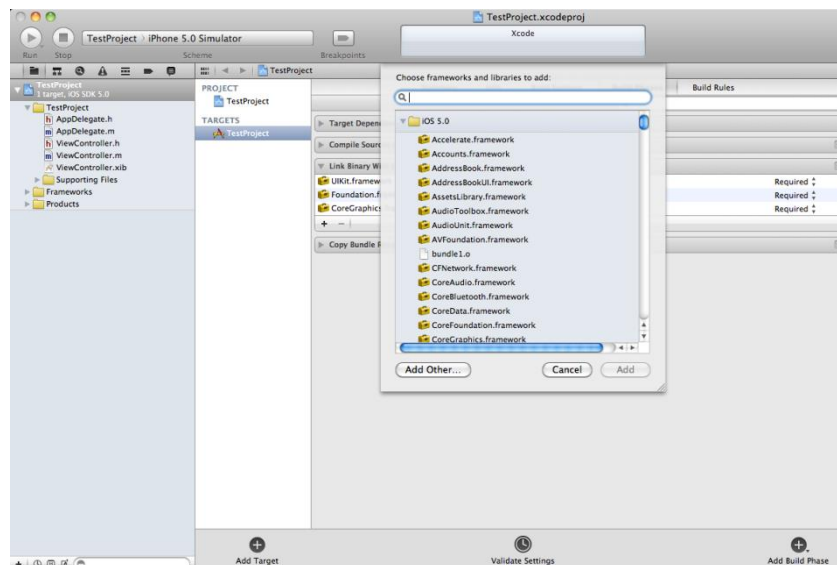
It is also recommended to keep the framework folder archived in a zip format, as it is delivered, until it is copied on a Mac development environment. Archiving the folder preserves the file attributes when it moves between different file-systems.

### 3\_2 Installation

The installation of framework provided by the PCL Add-on into an iOS Xcode project is straightforward and can be done by following few steps:

- Open/Create your iOS project (may be an application or cocoa touch library project)
- Select the target that will require the iSMP framework
- Go to the **“Build Phases”** menu
- Select and develop the **“Link Binary With Libraries”** build step
- Click on the **“+”** button, then the **“Add Other...”** button
- Navigate to your framework location and select its root folder **“iSMP.framework”**





This will add the framework to your project, configure the compiler to link the framework with the final application and search for the headers inside the **“Headers”** folder of the framework. Importing the framework to a target will set its **“Framework Search Path”** parameter to the actual path from which the framework was imported. To check the value of this parameter, do as following:

- Select the target to which the framework was added
- Go to the **“Build Settings”** menu
- Go to the **“Search Paths”** section
- Check the value of **“Framework Search Path”**

Please note that the instructions provided above require Xcode version 4.0 and higher. The procedure should be slightly different on older versions.

The library is now part of the project, and its headers can be included anywhere in the source files by writing the following line of code:

```
#import <iSMP/iSMP.h> // Include all headers
Or      #import <iSMP/HeaderName> // Include a specific header
```

The **“iSMP.h”** file imports the most used library header files but not all of them.

It is also convenient to put the import directives mentioned above in the precompiled header file of the Xcode project. This will make them visible for all the other source files within the project, and it won't then be necessary to import them elsewhere.

### 3\_3 Dependencies

The iSMP framework requires some other frameworks from the iPhone SDK to be linked with the target. When these libraries are missing, there will be some linking issues when building the project. Those are mentioned in the **“iSMP.h”** header file and include:

- Foundation,
- UIKit,
- CoreGraphics,
- SystemConfiguration,
- ExternalAccessory,

- CFNetwork.

The first 3 are mostly present in each iOS project. To add the missing ones, do as following:

- Select the target to which add the frameworks,
- Go to the “**Build Phases**” menu,
- Select and develop the “**Link Binary With Libraries**” build step,
- Click on the “+” button,
- Choose all the required libraries,
- And then click on the “**Add**” button.

## 3\_4 Naming Convention

One important naming convention used by the iSMP framework consists in prefixing all the data structures (Classes, C structures, Objective-C protocols) by the “**IC**” prefix that stands for “**Ingenico Companion**”. This helps taking advantage of Xcode’s code sense and auto-completion features.



iSMP framework versions prior to **3.2** do all use “**ICDevice**” as the base class for all the other classes handling the communication with the Companion. This class has been renamed to “**ICISMPDevice**” starting from the **3.2** release because it does conflict another class with the same name from Apple’s “**ImageCapture**” framework. This framework is private and invoked on certain situations when an application uses the camera or opens the photo library. A warning is printed on the iOS traces when the two classes are loaded at the same time, and this caused the communication with the Companion to be interrupted. The protocol “**ICDeviceDelegate**” associated with “**ICDevice**” has also been renamed to “**ICISMPDeviceDelegate**” to match the new class name.

The iSMP framework has also another naming rule used with protocols. A protocol defines a set of callback methods (events) and is associated with a class that fires those events. To highlight this association, in the iSMP library, each protocol name is composed of the name of the class with which the protocol is associated and a “**Delegate**” suffix. See examples below.

Example:

- The delegate of “ICAdministration” should implement the “ICAdministrationDelegate” protocol
- The delegate of “ICBarcodeReader” should implement the “ICBarcodeReaderDelegate” protocol
- The delegate of the abstract class “ICISMPDevice” should implement the “ICDeviceDelegate” protocol
- “ICAdministration” and “ICBarcodeReader” are both sub-classes of “ICISMPDevice”, so their delegate should also implement their super-class’s delegate protocol, which is “ICISMPDeviceDelegate”
- “ICTransaction” and “ICSPP” are both sub-classes of “ICISMPDeviceExtension” class, so, their delegate must conform to the “ICISMPDeviceExtensionDelegate” protocol.

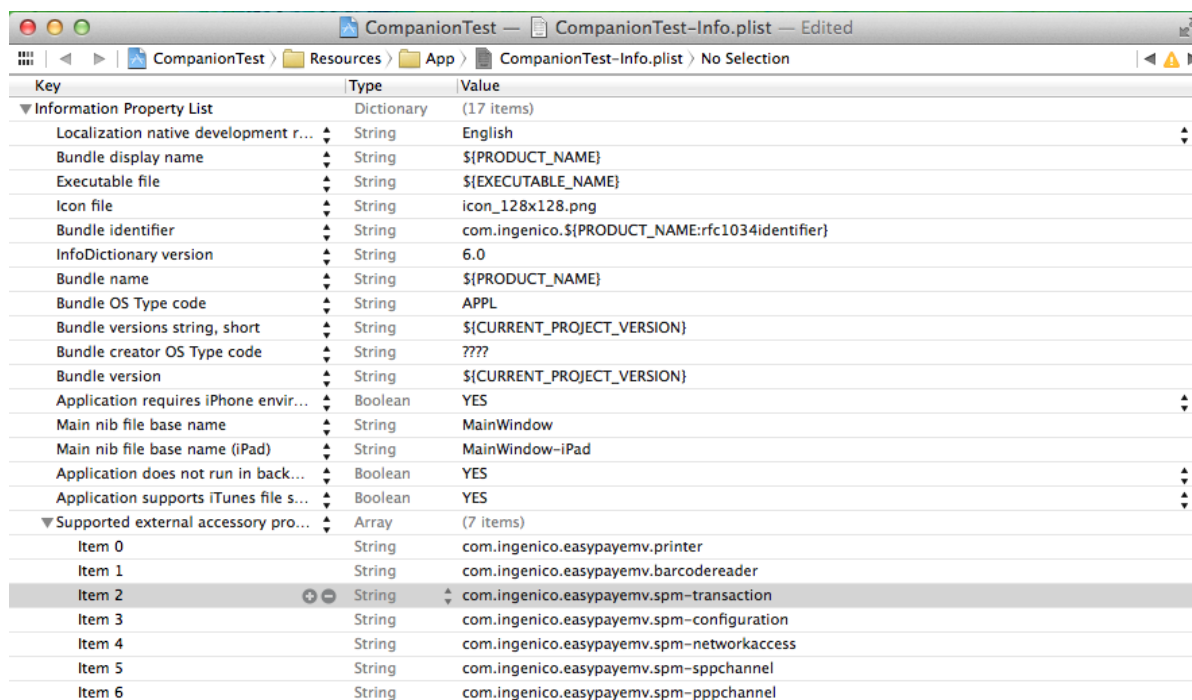
## 3\_5 Application Requirements

An application built with the iSMP framework must declare the protocol names that are used for the communication with the Companion device, within its manifest file (named “**app\_name-Info.plist**”).

The protocols supported by the Companion are:

- com.ingenico.easypayemv.printer
- com.ingenico.easypayemv.barcodereader
- com.ingenico.easypayemv.spm-transaction
- com.ingenico.easypayemv.spm-configuration
- com.ingenico.easypayemv.spm-networkaccess
- com.ingenico.easypayemv.spm-sppchannel
- com.ingenico.easypayemv.spm-pppchannel

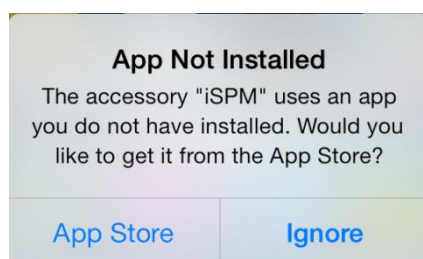
Those must be added to the file under the key “**Supported external accessory protocols**” as shown below:



Key	Type	Value
▼ Information Property List	Dictionary	(17 items)
Localization native development r...	String	English
Bundle display name	String	\$(PRODUCT_NAME)
Executable file	String	\$(EXECUTABLE_NAME)
Icon file	String	icon_128x128.png
Bundle identifier	String	com.ingenico.\$(PRODUCT_NAME:rfc1034identifier)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	\$(CURRENT_PROJECT_VERSION)
Bundle creator OS Type code	String	???
Bundle version	String	\$(CURRENT_PROJECT_VERSION)
Application requires iPhone envir...	Boolean	YES
Main nib file base name	String	MainWindow
Main nib file base name (iPad)	String	MainWindow-iPad
Application does not run in back...	Boolean	YES
Application supports iTunes file s...	Boolean	YES
▼ Supported external accessory pro...	Array	(7 items)
Item 0	String	com.ingenico.easypaymv.printer
Item 1	String	com.ingenico.easypaymv.barcodereader
Item 2	String	com.ingenico.easypaymv.spm-transaction
Item 3	String	com.ingenico.easypaymv.spm-configuration
Item 4	String	com.ingenico.easypaymv.spm-networkaccess
Item 5	String	com.ingenico.easypaymv.spm-sppchannel
Item 6	String	com.ingenico.easypaymv.spm-pppchannel



It is possible to declare only the protocols which are used by your application. But if you don't declare all of them, a warning message will appear the first time an Ingenico device will be connected to it. If you want to avoid this pop-up, declare all the protocols.



## 3\_6 Using the 64-bit version of the PCL framework

Starting with Add-on PCL for iOS 2.10, it is possible to develop either 32-bit or 64-bit iOS application. In a near future, Apple policy will be to reject 32-bit only applications, so all your application must be migrated to support 64-bit. To create an application that supports both architectures, follow the below steps:

- Install Xcode 5.0.1 or higher

- Open your project. Xcode prompts you to modernize your project. Modernizing the project adds new warnings and errors that are important when compiling your app for 64-bit.
- Update your project settings to support iOS 5.1.1 or later. You can't build a 64-bit project if it targets an iOS version earlier than iOS 5.1.
- Change the Architectures build setting in your project to "Standard Architectures (including 64-bit)."
- Update your app to support the 64-bit runtime environment. The new compiler warnings and errors will help guide you through this process. However, the compiler doesn't do all of the work for you; use the information in this document to help guide you through investigating your own code.
- Test your app on actual 64-bit hardware. iOS Simulator can also be helpful during development, but some changes, such as the function calling conventions, are visible only when your app is running on a device.
- Use Instruments to tune your app's memory performance.
- Submit an app that includes both architectures for approval.
- 

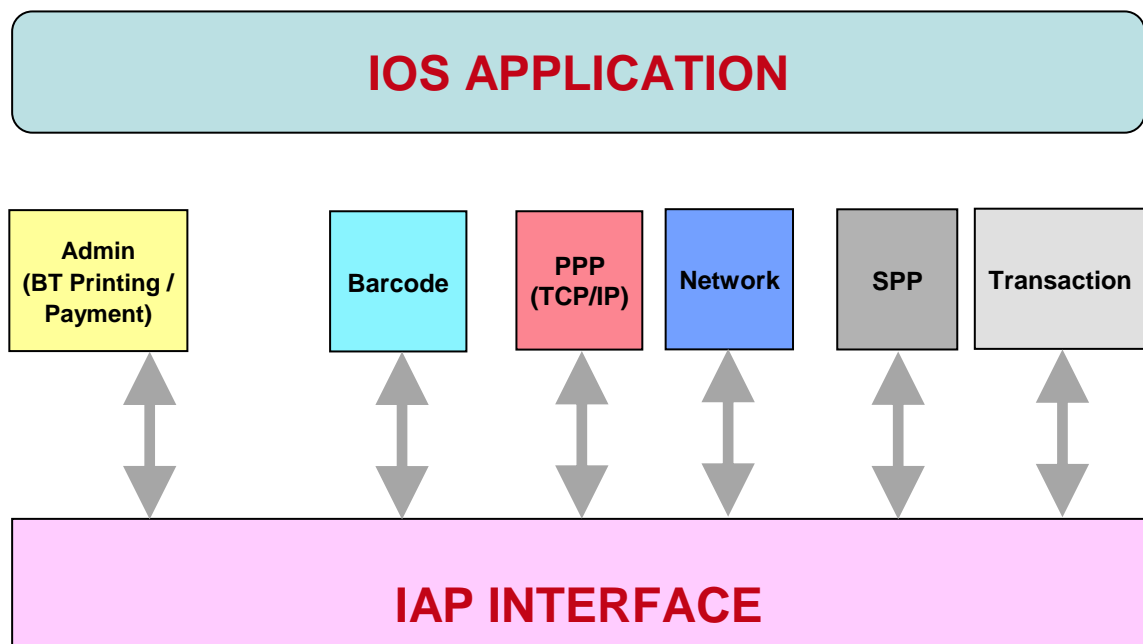
More information about migration can be found in Apple's document [CocaTouch64bitGuide.pdf](#)

## 4 Developing your application

### 4\_1 Architecture

The PCL add-on offers different communication channels, each of which has a specific use. Those include:

- Barcode Channel
- Administration Channel (Standalone payment or BT printing)
- Network Channel (this channel is no longer to be used and will be deprecated in the future versions)
- Transaction channel
- Transparent SPP Channel
- PPP Channel (provide a native TCP/IP connection between iOS and Telium)



The PCL add-on provides a set of API to access those services and interact with the Companion through properties, methods and callbacks that will be detailed throughout this document.

### 4\_2 General principles

#### 4\_2\_1 Delegate

Delegation is a simple and powerful pattern in which one object in a program acts on behalf of, or in coordination with, another object. The delegating object keeps a reference to the other object (the delegate) and at the appropriate time sends a message to it. The message informs the delegate of an event that the delegating object is about to handle or has just handled. The delegate may respond to

the message by updating the appearance or state of itself or other objects in the application, and in some cases it can return a value that affects how an impending event is handled. The main value of delegation is that it allows you to easily customize the behavior of several objects in one central object.

Please refer to the [Erreur ! Source du renvoi introuvable.] documentation to know what delegate functions must be implemented for each class.

## 4\_2\_2 Additional Compiler Flags When Using Categories

The “**ICAdministration**” class comes with different categories:

- “**StandAlone**” category: regroups the methods used for payment
- “**iBP**” category: regroups the methods used to print document from the iOS device on a Bluetooth printer connected to the Companion.

Applications linked against the iSMP framework **must add extra compiler flags to support the Categories introduced by the Objective-C language**. These flags are:

- “**-ObjC**”: this flag is always required
- “**-all\_load**”: this flag is required for old Xcode versions that had a compiler issue. To be sure, just put them both since there was no communication from Apple on when the issue was fixed.

## 4\_2\_3 Managing background state



It is mandatory to do a powerOff of the barcode reader (see the section related to the barcode reader below) when your business application goes to background. The powerOff function must be called in the applicationDidEnterBackground callback of your application.

## 4\_3 Checking the connection state between the device and the Companion

The “**ICISMPDevice**” class has properties and methods that provide information about a communication channel and help keep track of the state of the connection between the iOS device and the Companion.

Among those, the following class method checks if the Companion and the iOS device are connected: **(BOOL)isAvailable**

If the iOS device is linked with the Companion and the authentication between the two devices went well (the connection state icon on the Companion screen header is full: ■), this method return **YES**, otherwise it returns **NO**.

The other properties are inherited by the derived classes, mentioned previously, and should be called on instances of those classes.

Name	Type	Description
<b>(BOOL)isAvailable</b>	Property	Indicates if a channel is open and ready
<b>(NSString *)protocolName</b>	Property	Returns the protocol name of a certain channel
<b>(NSInputStream *)inStream</b>	Property	Returns a pointer to the input stream used to read data from the Companion on a given channel

<b>(NSOutputStream *)outStream</b>	Property	Returns a pointer to the output stream used to write data to the Companion on a given channel
------------------------------------	----------	---

The most important property among those is ***isAvailable***. This property must be tested by the application after initializing a communication session. The returned value determines whether the session is open and that the Companion is ready to exchange data and reply to requests.

The ***inStream*** and ***outStream*** properties can be useful to debug communication issues and get the state of the serial streams opened to the Companion, although it is very unlikely to experience such issues. Those properties are however essential for communication on the **Transaction** channel that does not, like the four others, provide a high level API to write commands to the Companion, but provides instead raw read/write streams that can be used to send customized commands or to implement a customized communication protocol between the two devices. This part will be detailed in “**ICTransaction**” class.

## Connection State Changes

It is possible and recommended to keep track of connection state changes after opening a communication session. Indeed, the connection between the two devices may be lost unpredictably. This may happen at different situations:

- The two devices are unlinked (Bluetooth connection broken or iOS device detached in case of iSMP)
- The Companion reboots to apply an update
- The iOS device goes into sleep mode and closes the communication channels

During such scenarios, the iOS application and the iSMP library objects that handle the communication remain alive. They should then be able to detect the next time that the Companion is available to re-establish the communication if necessary. This can be done by implementing the two following methods defined within the “**ICISMPDeviceDelegate**” protocol:

Callback	Calling Context
<b>(void)accessoryDidConnect:(ICISMPDevice *)sender</b>	Main Runloop
<b>(void)accessoryDidDisconnect:(ICISMPDevice *)sender</b>	Main Runloop

Those two methods are called on the delegate of an instance of “**ICISMPDevice**” to notify it with Companion’s connection and disconnection events. The “**sender**” argument is a pointer to the source of the event, which is a pointer to an “**ICISMPDevice**” object. The “**sender**” parameter is important in case the delegate implements different protocols at the same time (for example “**ICPrinterDelegate**” and “**ICNetworkDelegate**”). In this case, it becomes necessary to be able to distinguish which channel has connected or disconnected.

The two callbacks are both called on the main run-loop of the program. This means that it is safe to manipulate User Interface objects where they are implemented.



**accessoryDidConnect** callback may be called at different times depending on the class that triggered it. This may happen either before or after a communication channel is opened:

The callback is called to signal the presence of the Companion, which means that the application can open the communication session. Examples of classes that have this behavior are “**ICBarcodeReader**” and “**ICAdministration**”.

The callback is called to signal that the communication channel associated with the **sender** object provided as argument has been opened and ready. “**ICPrinter**”, “**ICSP**”, “**ICNetwork**”, “**ICPPP**” and



“**ICTransaction**” classes have this behavior. These classes open the communication session at initialization and each time the external accessory (Companion) is detected and close it when they are released.

---

## 4\_4 Selecting the Telium device to use

Before starting using the various channels, you should specify which Telium device you want to use. This is achieved by the use of two functions:

- `ICISMPDevice getConnectedTerminals`, that provides the list of currently connected Telium terminals to the iOS device. Note that “connected” refers to the Bluetooth connection and not just to the Bluetooth pairing.
- `ICISMPDevice setWantedDevice`, that specifies which Telium device you want to use, from the list of connected ones.

It will be up to your application to code the display of the connected terminal and get which one has been selected to use.

Depending on your application architecture, it is most likely that the final user will be able to enter the page which the device selection many time while the application is running (for instance, if it has been included in a parameters page). To be sure that the page provides consistent information, you can use the `ICISMPDevice getWantedDevice` function to retrieve which device is currently in use and update the page accordingly.

Refer to the `CompanionTestSample` (`CompanionSelectionViewController.m` file) project to see how this can be implemented.

If you want to keep the selected device even after the application has been closed or iOS has restarted, you will need to save the Device information in the application properties, you can use for instance the code below:

```
NSString *selectedCompanionToSave = [[ICISMPDevice getConnectedTerminals]
objectAtIndex:indexPath.row];
[ICISMPDevice setWantedDevice: selectedCompanionToSave];
NSLog(@"Companion selected is %@", selectedCompanionToSave);
NSUserDefaults *prefs = [NSUserDefaults standardUserDefaults];
[prefs setObject:selectedCompanionToSave forKey:@"IngenicoCompanionInUse"];
```

It is possible to not specify which device to use, if you are absolutely sure than you will never have many Telium connected at once with the same iOS device. If the device selection is not done at the application level, PCL will randomly choose one Telium device from the list.



In case of you don't specify the device to use with the function `setWantedDevice` you can have faulse connect/disconnect. If you use another device that have an IAP chipet like a BT Printer for example, the callback `onAccessoryConnect/onAccessoryDisconnect` can be call when this printer is connected or disconnected. To avoid these disagreements it necessary to set the device used with the function `setWantedDevice`.

---



## 4\_5 Doing a payment

### 4\_5\_1 Sending the payment request

To start a payment transaction, two functions might be called:

Method	Type
<b>(void)doTransaction:(ICTransactionRequest)request</b>	Asynchronous
<b>(void)doTransaction:(ICTransactionRequest)request withData:(NSData *)extendedData andApplicationNumber:(NSInteger)appNum</b>	Asynchronous

Both functions are asynchronous, meaning they return immediately and the result comes later. They have a default timeout value set to 60 seconds. The timeout can be changed in the iOS side using the following methods, although it is not recommended to change it unless for testing purposes.

Method	Description
<b>(void)setDoTransactionTimeout:(NSInteger)timeout</b>	Set the timeout of Do Transaction
<b>(NSInteger)getDoTransactionTimeout</b>	Get the timeout of Do Transaction

“doTransaction:” and “doTransaction:withData:andApplicationNumber:” do pretty much the same. The second comes with extra parameters that are used to pass additional data to the terminal and select the application that should handle the transaction. Calling “doTransaction” is equivalent to calling “doTransaction:withData:andApplicationNumber:” with a null data and zero application number.

The additional data that is passed through the “extendedData” argument has not a predefined format. iOS application developers and Telium application developers should agree on the format of the data to be exchanged.

Both “doTransaction” methods have however a common input data structure of type “ICTransactionRequest”. This has to be filled with all the required information for the transaction. The descriptions of the fields contained within this structure are in the table just below:

Field	Description
<b>posNumber</b>	POS Number
<b>amount</b>	Amount of the transaction
<b>accountType</b>	Account Type
<b>transactionType</b>	Transaction Type (debit, credit, ...)
<b>currency</b>	Currency (“EUR”, “USD”, ...)
<b>privateData</b>	Application Specific Data
<b>delay</b>	Indicates if the transaction is processed synchronously or asynchronously on the terminal side – Since LibiSMP_v3.2, this field is set internally to ‘0’ by the iSMP library.
<b>authorization</b>	Authorization flag

### Transaction Types

The transaction types supported by the “doTransaction” methods are listed in the table below:

Constant	Description
<b>‘0’</b>	Debit
<b>‘1’</b>	Credit
<b>‘2’</b>	Cancellation
<b>‘3’</b>	Duplicata
<b>‘4’-‘9’</b>	Reserved for future use

The “transactionType” field of “ICTransactionRequest” should be set to one of the aforementioned constants.

### Handling the Result

The result of “Do Transaction” is provided asynchronously to the iOS application through two callbacks defined within the “ICAdministrationStandAloneDelegate” protocol.

Callback	Calling Context
<b>(void)transactionDidEndWithTimeoutFlag:(BOOL)replyReceived</b>  <b>result:(ICTransactionReply)transactionReply</b> <b>andData:(NSData *)extendedData</b>	Main Runloop

The callback provides an additional “extendedData” parameter that contains the extra data returned by the payment application when the transaction is performed in extended mode. This field is null if the transaction does not use extra data.

The table below lists the arguments of this callback and their significance.

Argument	Description
<b>replyReceived</b>	Timeout flag ( <b>YES</b> : transaction processed, <b>NO</b> : transaction timeout)
<b>transactionReply</b>	Outcome of the transaction contained within a “ICTransactionReply” structure
<b>extendedData</b>	Extra Returned Data

The “ICTransactionReply” structure contains the output of the basic transaction and has the following fields:

Field	Description
<b>posNumber</b>	POS Number (same as in the request)
<b>operationStatus</b>	Status of the “Do Transaction” request
<b>amount</b>	Amount of the transaction
<b>accountType</b>	Account Type
<b>currency</b>	Currency (same as in the request)

<b>privateData</b>	Specific payment application data
<b>zoneRep</b>	Response of the cash register connection
<b>zonePriv</b>	Private area

When implementing the callback, the iOS application should do the following:

- Check the “**replyReceived**” argument to know if the transaction was processed or if it failed because of a timeout.
- Check the “**operationStatus**” field of “**transactionReply**” to determine the result of the transaction. The possible results are listed in the table below:

Result	Description
<b>'0'</b>	Operation successful
<b>'7'</b>	Operation failed

## 4\_5\_2 Capturing the signature

During a transaction, the Companion may request the signature of the customer. This information is provided to the iOS POS application through callbacks defined within the “**ICAdministrationStandAloneDelegate**” protocol.

Callback	Calling Context
<b>(void)shouldDoSignatureCapture:(ICSignatureData)signatureData</b>	Main Runloop
<b>(void)signatureTimeoutExceeded</b>	Main Runloop

“**shouldDoSignatureCapture**” is called on the application’s main thread when a signature capture request is received. This callback provides information about the size of the signature and the timeout before which it should be captured and submitted to the terminal. This information is contained within the “**signatureData**” argument that has the following fields:

Field	Description
<b>screenWidth</b>	Width of the signature
<b>screenHeight</b>	Height of the signature
<b>userSignTimeout</b>	Signing timeout

The “**screenWidth**” and “**screenHeight**” are important since they specify the size of the signature that should be provided back to the terminal. The application must build a “**UIImage**” object out of the captured signature, and provide it to the terminal by calling the following method:

**-(BOOL)submitSignatureWithImage:(UIImage \*)image;**

This method sends the signature bitmap to the terminal. It is synchronous and returns immediately. It must be called before reaching the signing timeout that is equal to the value of “**userSignTimeout**” (this expiration of the timeout is signaled by the “**signatureTimeoutExceeded**” callback). The result of the call is a Boolean that indicates whether the submission succeeded or failed.

The “**image**” object may be of any iOS supported bitmap configuration (RGB, gray scale, 24 bits or 16 bits encoded ...). It undergoes, when submitted, some transformations to convert it into a black and white monochrome bitmap, where each pixel is encoded using 1 bit. This final configuration is the one supported by the Companion and its compatible printers.

## 4\_5\_3 Printing the receipt

The “**ICAdministration**” class provides a printing approach to print receipts provided by the Companion as text strings and bitmaps. In the contrary to the printer channel (native Companion printing) that will be discussed in a later section, the advantage of this technique is that all the formatting and rendering of the receipt is done on the iOS side, which makes it quicker than the native printing especially when the size of the receipt is relatively big.

Printing using this approach is performed sequentially. The application receives the printing requests issued by the Companion through the adequate callbacks. It is then free to render the receipt accordingly. The choice of the rendering technique is up to the application developer. There are no

constraints in this regard. The application can even do the printing on an external Made For iPhone Bluetooth printer.

When the terminal has a ticket to be printed and rendered on the iOS side, it provides the elements composing this ticket (text, bitmaps, formatting styles, to the iOS app using the callbacks received by **ICAdministration** and defined within the “**ICAdministrationStandAloneDelegate**” protocol. These callbacks are listed in the table below:

Callback	Calling Context
<b>(void)shouldPrintText:(NSString*)text withFont:(UIFont*)font andAlignment:(UITextAlignment)alignment</b>	Main Runloop
<b>(void)shouldPrintText:(NSString*)text withFont:(UIFont*)font alignment:(UITextAlignment)alignment XScaling:(NSInteger)xFactor YScaling:(NSInteger)yFactor underline:(BOOL)underline</b>	Main Runloop
<b>(void)shouldPrintText:(NSString*)text withFont:(UIFont*)font alignment:(UITextAlignment)alignment XScaling:(NSInteger)xFactor YScaling:(NSInteger)yFactor underline:(BOOL)underline bold:(BOOL)bold</b>	Main Runloop
<b>(void)shouldPrintRawText:(char*)text withCharset:(NSInteger)charset withFont:(UIFont*)font alignment:(UITextAlignment)alignment XScaling:(NSInteger)xFactor YScaling:(NSInteger)yFactor underline:(BOOL)underline bold:(BOOL)bold</b>	Main RunLoop
<b>(void)shouldPrintImage:(UIImage*)image</b>	Main Runloop
<b>(void)shouldFeedPaperWithLines:(NSUInteger)lines</b>	Main Runloop
<b>(void)shouldCutPaper</b>	Main Runloop
<b>(NSInteger)shouldStartReceipt:(NSInteger)type</b>	Main RunLoop
<b>(NSInteger)shouldEndReceipt</b>	Main RunLoop
<b>(NSInteger)shouldAddSignature</b>	Main RunLoop

#### 4\_5\_3\_1 Printing Text

The callback that informs the iOS app of the presence of text strings to be printed and how they should be rendered on the final receipt is defined as below:

```

-(void)shouldPrintText:(NSString*)textwithFont:(UIFont*)font
alignment:(UITextAlignment)alignment
        XScaling:(NSInteger)xFactor YScaling:(NSInteger)yFacotor
        underline:(BOOL)underline
        bold:(BOOL)bold;

```

Parameter	Description
<b>Text</b>	The text string to be printed
<b>Font</b>	“ <b>UIFont</b> ” object that specifies the font to be used. This can be retrieved through its “ <b>fontName</b> ” method
<b>Alignment</b>	A text alignment constant. Possible alignments are left, center or right
<b>XScaling</b>	The factor by which the text should be scaled in the X direction. Possible values are 1, 2 or 4 (normal, double or quadruple text width)
<b>YScaling</b>	The factor by which the text should be scaled in the Y direction. Possible values are 1, 2 or 4 (normal, double or quadruple text height)

<b>Underline</b>	Specifies whether the text provided should be underlined or not
<b>Bold</b>	Specified whether the text should be normal or bold

The size of the font is however not specified by the terminal application. It is up to the iOS developer to choose a convenient font size for text of the receipt. The Companion application can however ask that the size of the text has to be doubled or quadrupled by changing the “**XScaling**” and “**YScaling**” factors.

The names of the text fonts supported by iOS are mapped to integer IDs that are set by the Companion application when it issues a text printing request. This table lists all supported fonts and their respective IDs.

ID	Font	ID	Font
0	@"Baskerville"	29	@"Helvetica-BoldOblique"
1	@"AmericanTypewriter"	30	@"HelveticaNeue"
2	@"AmericanTypewriter-Bold"	31	@"HelveticaNeue-Bold"
3	@"AppleGothic"	32	@"STHeitiSC-Light"
4	@"ArialMT"	33	@"STHeitiSC-Medium"
5	@"Arial-ItalicMT"	34	@"STHeitiTC-Light"
6	@"Arial-BoldMT"	35	@"STHeitiTC-Medium"
7	@"Arial-BoldItalicMT"	36	@"STHeitiJ-Light"
8	@"ArialHebrew"	37	@"STHeitiJ-Medium"
9	@"ArialHebrew-Bold"	38	@"STHeitiK-Light"
10	@"ArialRoundedMTBold"	39	@"STHeitiK-Medium"
11	@"Courier"	40	@"HiraKakuProN-W3"
12	@"Courier-Oblique"	41	@"HiraKakuProN-W6"
13	@"Courier-Bold"	42	@"MarkerFelt-Thin"
14	@"Courier-BoldOblique"	43	@"Thonburi"
15	@"CourierNewPSMT"	44	@"Thonburi-Bold"
16	@"CourierNewPS-ItalicMT"	45	@"TimesNewRomanPSMT"
17	@"CourierNewPS-BoldMT"	46	@"TimesNewRomanPS-ItalicMT"
18	@"CourierNewPS-BoldItalicMT"	47	@"TimesNewRomanPS-BoldMT"
19	@"DBLCDTempBlack"	48	@"TimesNewRomanPS-BoldItalicMT"
20	@"GeezaPro"	49	@"TrebuchetMS"
21	@"GeezaPro-Bold"	50	@"TrebuchetMS-Italic"
22	@"Georgia"	51	@"TrebuchetMS-Bold"
23	@"Georgia-Italic"	52	@"Trebuchet-BoldItalic"
24	@"Georgia-Bold"	53	@"Verdana"
25	@"Georgia-BoldItalic"	54	@"Verdana-Italic"
26	@"Helvetica"	55	@"Verdana-Bold"
27	@"Helvetica-Oblique"	56	@"Verdana-BoldItalic"
28	@"Helvetica-Bold"	57	@"Zapinfo"

**-(void)shouldPrintRawText:(char\*)text withCharset:(NSInteger)charset  
withFont:(UIFont\*)font alignment:(UITextAlignment)alignment  
XScaling:(NSInteger)xFactor YScaling:(NSInteger)yFactor  
underline:(BOOL)underline  
bold:(BOOL)bold;**

shouldPrintRawText, tells the application that some text must be printed with a character set defined by the Telium application. An example is provided in CompanionTestSample application (Administration > A013 – Printing).

### 4\_5\_3\_2 Printing Images

Companion applications can request the “**ICAdministration**” object to print a “**UIImage**” object. This will fire the “**shouldPrintImage**” callback on the delegate with the parameter “**image**” which is the image object. The image is in black and white, and has a maximum width of 384 point.

### 4\_5\_3\_3 Feeding Paper

This callback asks the application to feed paper if it is printing on an external printer. If the application is rendering the receipt locally into a bitmap or a PDF context, it has just to skip one or some lines.

### 4\_5\_3\_4 End of Printing

The callback “**shouldCutPaper**” signals the end of printing. This callback instructs the iOS application to cut the paper on the printer when the receipt is ready.

## 4\_5\_4 Exchanging additional information with the payment application

The send message API is used to exchange 1024 bytes buffers between the iOS device and the Companion. This mechanism is based on the following method defined within the “**StandAlone**” category of “**ICAdministration**”.

Method	Type
<b>(BOOL)sendMessage:(NSData*)data</b>	Synchronous

The result is a Boolean value that tells whether the message was delivered or not to the terminal.

Companion messages arrive to the iOS application through the following callback defined within the “**ICAdministrationStandAloneDelegate**” protocol:

Callback	Calling Context
<b>(void)messageReceivedWithData:(NSData *)data</b>	Main Runloop



Since the callback is performed on the main thread of the application, the “**sendMessage**” function can be called inside “**messageReceivedWithData:**”. It is however better to perform “**sendMessage**” in a background thread since it has a 3 second timeout.

## 4\_6 Managing the terminal

“**ICAdministration**” class provides all the API to manage the Companion, please refer to doxygen documentation for more information about class initialization.

### 4\_6\_1 Terminal information

The “**ICISMPDevice**” class has many methods that return information about the Companion. This information requires however the terminal to be plugged and connected to the iOS device since it is the result of the authentication process.

Method	Description
<b>(NSString *)serialNumber</b>	Returns the last 8 digits of the serial number. The full serial number can be retrieved using the <code>getFullSerialNumber</code> function (see section below)
<b>(NSString *)modelName</b>	Returns the model number composed of twelve digits
<b>(NSString *)firmwareRevision</b>	Returns a string representation of the Companion's firmware revision
<b>(NSString *)hardwareRevision</b>	Returns a string representation of the Companion's hardware revision
<b>(NSString *)name</b>	Returns the string “iSPM”

Those methods return a null string when the iOS device and the Companion are not connected.

### 4\_6\_2 Terminal management

The **ICAdministration** class gives you access to various functions to manage your terminal

Method	Usage	Type
<b>(BOOL)setDate</b>	This API sets the time/date on the Companion to that of the iOS device	Synchronous
<b>(NSDate *)getDate</b>	This request gets the system date and time of the Companion.	Synchronous
<b>(BOOL)isIdle</b>	This method checks the state of the Companion, whether it is idle or busy doing some tasks.	Synchronous
<b>(ICDeviceInformation)getInformation</b>	This function returns information about the Companion, including its serial number, reference code	Synchronous
<b>(NSString*)getFullSerialNumber</b>	This function returns the full serial number of the Companion.	Synchronous
<b>(void)reset:(NSUInteger)resetInfo</b>	This function asks the Companion terminal to reboot	Asynchronous



<b>(BOOL)simulateKey:(NSUInteger)key</b>	<p>This method simulates key input on the Companion. This API has to be used in an intelligent manner to automate some actions that require navigating through the menus of the Companion. The iOS application can then simulate a series of key that start a certain action or that configure the Companion.</p> <p>The iOS device has, however, no idea about the configuration in which the Companion is set. Therefore, the simulation must be performed by knowing in advance which and how many menus are available.</p>	Synchronous
<b>(NSArray *)getSoftwareComponents</b>	This function returns all the components loaded into the Companion	Synchronous
<b>(NSString *)getSpmciVersion</b>	This function returns the version of the SPMCI Telium component	Synchronous
<b>(ICTmsInformation *)getTmsInformation</b>	This function returns the parameters of the TMS server on TELIUM side. This includes the IP address or hostname, port, SSL certificate selected and the list of certificates installed	Synchronous
<b>(ICTmsInformation *)setTmsInformation</b>	This function permits to set each parameter of the TMS server on TELIUM side	Synchronous



When using the `setDate` function, if the time difference between the iOS device and the Companion is greater than 2 days, the request fails.

### 4\_6\_3 Remote download

This function starts the download process in order to update the software components loaded into the Companion. When this process is started, the terminal will try to connect to the Terminal Management System (T.M.S.) which is the server that keeps track of terminal updates and provides the files to download. The parameters necessary to connect to the TMS are already preconfigured in the terminal and do not need to be provided by the iOS application.



When connecting to the TMS, the terminal needs to have access to network. This means that the “`ICNetwork`” or “`ICPPP`” class must be instantiated in order to route the Companion connection

through the iOS device. The latter has also to be connected to a network (using Wi-Fi or 3G) where the TMS is reachable.

The function that starts the download is given in the following table:

Method	Type
<b>(BOOL)startRemoteDownload</b>	Synchronous

This function is synchronous and returns when the download is done or when the timeout is reached. This method returns YES when the download is fulfilled, and NO when it fails.

**The Companion always reboots after returning from “startRemoteDownload”, even if the download fails.**

## 4\_6\_4 Power Management



When the iSMP is used (physical integration between Ingenico device and iOS device), the **sleep modes of the terminal and the iOS device (iPod Touch, iPad or iPhone) are synchronized**. The iOS device is the master, which means that when the iOS goes to sleep mode (screen locked), it notifies the Ingenico device, that goes to standby mode just after. This no longer applies when the iOS device is unplugged from the iSMP

### 4\_6\_4\_1 Backlight & Suspend Timeouts



This section does not apply to iSMP devices as standby parameters are overridden by iOS device behavior. However, if you unplug the iOS device, these parameters will be taken into account.

The backlight timeout is the inactivity time after which the Companion reduces the light of its screen. The suspend timeout is the inactivity time after which the Companion goes into auto sleep mode. As soon as the two devices are not synchronized (iPod not plugged or asleep), this parameter is taken into account.

The following functions/properties are used to set and get the values of the Companion backlight and suspend timeouts:

Method	Type
<b>(BOOL)setBacklightTimeout:(NSInteger)backlightTimeout andSuspendTimeout:(NSInteger)suspendTimeout</b>	Synchronous
<b>(NSInteger)backlightTimeout[read, assign]</b>	Synchronous
<b>(NSInteger)suspendTimeout[read, assign]</b>	Synchronous

The “**setBacklightTimeout:andSuspendTimeout:**” function sets the timeouts for backlight and auto sleep mode. The timeouts are provided as integer values and represent a time value in seconds. This function is synchronous and does only return when the request is executed by the Companion or when the timeout is reached. It returns **YES** when the call succeeds, and **NO** in the other case.

“**backlightTimeout**” and “**suspendTimeout**” are read-only and return the current backlight and timeout values configured into the Companion. When the request fails (Companion disconnected or any other reason), they return the value -1.

### 4\_6\_4\_2 Companion Battery Level

“**ICAdministration**” has a “**batteryLevel**” property that returns the current battery level of the Companion.

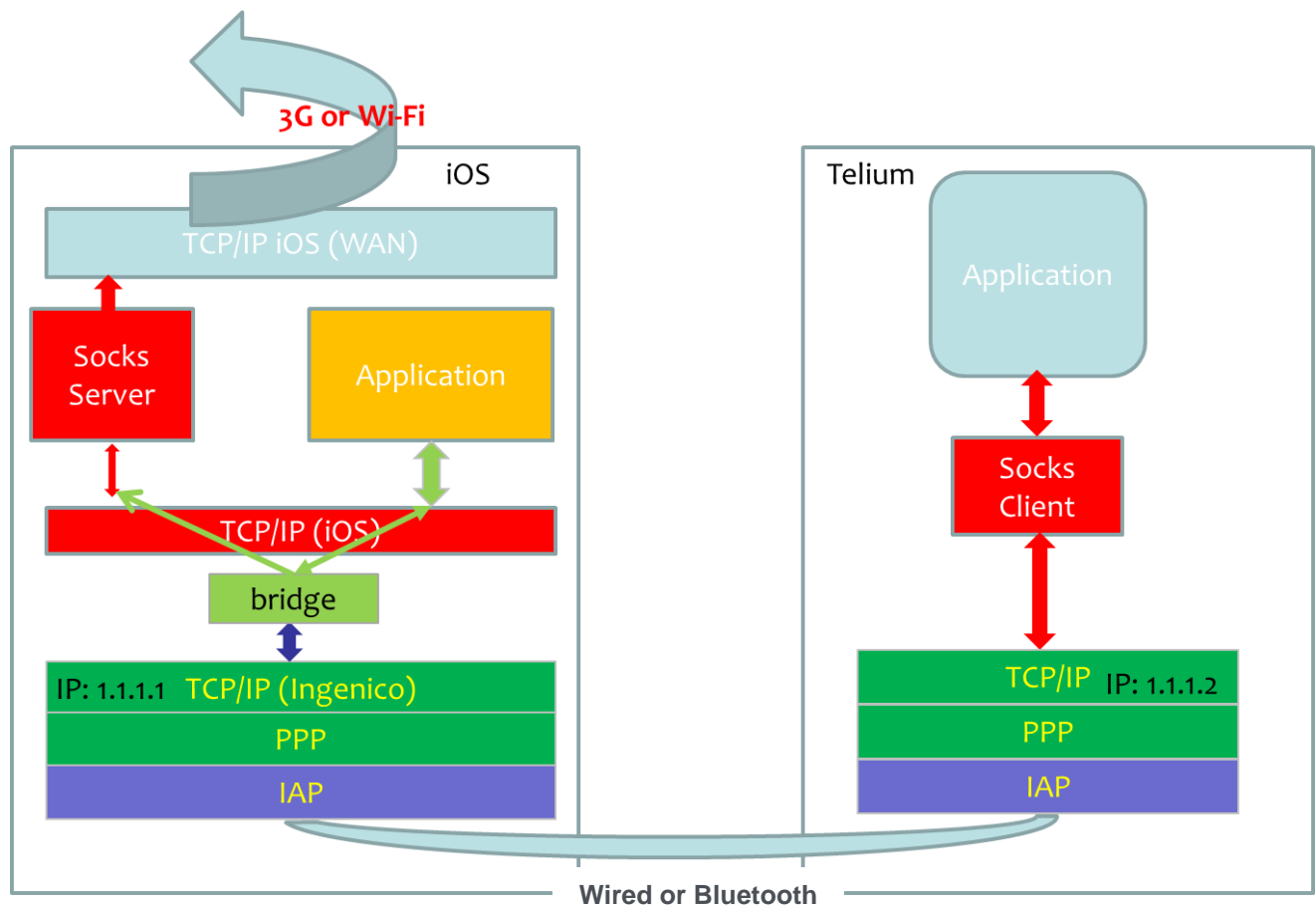
Method	Type
<b>(NSInteger)batteryLevel</b>	Synchronous

When the request is successful, the result is an integer value ranging from 0 to 100. If an error occurs, the function returns -1

## 4\_7 Using the network connection of the iOS device

The “ICPPP” class provides a native TCP/IP connection between your Telium device and the iOS device, thus allowing the use the iOS Internet connectivity by the Companion. This connection will allow TCP and SSL connection, in both directions.

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



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

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

The bridge must be configured before an iOS 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 iOS application: 1080.



When the iOS device and the Companion lose synchronization (due to Bluetooth or iAP disconnection), all the open connections and sockets are closed and will not be reopen automatically when the physical connection will be reestablished. It is up to the application to manage the loss of the physical connection.

## 4\_7\_1 Prerequisites

“**ICPPP**” requires “**ICAdministration**” to work. Indeed, the latter provides, behind the scenes, the available iOS device’s network interfaces to the Companion when asked for that. Therefore, it’s necessary to initialize “**ICAdministration**” too, when initializing “**ICPPP**”.

## 4\_7\_2 Connecting from iOS to Telium

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 `addiOSToTerminalBridgeOnPort` function, XXXX as integer value. For example:

```
//Open the port for incoming connection  
[self.pppChannel addiOSToTerminalBridgeOnPort:XXXX];
```

- On iOS, open the connection using the iOS localhost IP address (127.0.0.1 or localhost as you want) and the [XXXX] port number

Alternatively, you can use the `addiOSToTerminalBridgeLocalOnPort` function if you want, for security reason for instance, to restrict the call to the Companion coming only from the local iOS.

```
//Open the port for incoming connection  
[self.pppChannel addiOSToTerminalBridgeLocalOnPort:XXXX];
```

## 4\_7\_3 Connecting from Telium to iOS

In order to successfully call a TCP server located on iOS, the following actions must be performed:

- Have the TCP server listening on port [XXXX] on iOS
- Add the [XXXX] port value to the bridge configuration with `addTerminalToiOSBridgeOnPort` function, XXXX as integer value
- On Telium, open the connection using the IOS device IP address (1.1.1.1) and the [XXXX] port number

```
//Open the port for incoming connection  
[self.pppChannel addTerminalToiOSBridgeOnPort:XXXX];
```

## 4\_7\_4 Connecting from Telium to remote host through iOS socks server

To be able to use the communication features provided by the iOS 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



Then your Telium application can use standard TCP/IP functions to connect to remote host (either BSD or LinkLayer 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\_7\_5 Remote host connection state

The `setServerConnectionState` function of the class **ICAdministration** can be used to personalize the Telium header by changing the connectivity icon . Note that when using this function, the whole synchronization between the header and the connection status must be managed by your application. The table below shows the icon that will appear depending on state of the connection that is specified.

State	Icon
<b>Connected to Server</b>	
<b>No Connected to Server</b>	

## 4\_7\_6 ICPPPPDelegate Protocol

This protocol provides a set of callbacks to be implemented by the delegate of “**ICPPPP**”. These are listed in the table below and will be detailed one by one just after.

Callback	Calling Context
<b>(void) pppChannelDidOpen</b>	Random Runloop
<b>(void) ppp ChannelDidClose</b>	Random Runloop

- (void) ppp ChannelDidClose

This event is fired when the ICPPP channel is closed without calling `ICPPP_closeChannel`. This may happen when the terminal disconnects or when the serial link between the two devices is broken (when losing the Bluetooth connection or when unplugging the devices). After receiving this event, network communication is no longer possible until the channel is opened again by calling `ICPPP_openChannel`.

- (void) pppChannelDidOpen

ICPPP channel becomes ready when the TCP/IP stack is initialized and the terminal and iOS device are connected using the IP protocol. At this stage, the connection attributes can be retrieved through the properties: IP, submask, DNS and terminalIP.

When this event is fired, communication using the bridges to and from the terminal becomes possible.

For more information about all the callbacks please refer to the doxygen documentation.

## 4\_8 Using a generic COM port

By using the classes providing in the PCL add-on, you have access to a virtual COM port between the device and the Companion.

The **ICTransaction** class provides means to write and read data to and from a raw communication channel that works exactly like a serial port.

The transaction channel is accessible on the Telium side through the **COM0** port.

### 4\_8\_1 Synchronous Approach

The synchronous approach consists in writing as much data provided as possible to the output stream. If the data was not all written, the same function should be called again with the remaining data. It is convenient in this case, to implement a loop to keep calling the “**SendData:**” or “**SendString:**” until all the data is sent. Those methods return the number of written bytes, which can be used as an offset to send the remaining data.

While in the loop, it is necessary to check if the Companion device is still available and ready to receive the data. That’s why, it is important to call the “**isAvailable**” property of the subclass of “**ICISMPDeviceDelegate**” to make sure the channel is open.

### 4\_8\_2 Asynchronous Approach

An iOS application that would communicate over the transaction channel has two means of doing it:

- Using the “**inStream**” and “**outStream**” inherited from “**ICISMPDevice**” class: those are raw streams created by “**ICISMPDevice**” when the transaction accessory session is opened.
- Using the send/receive API inherited from “**ICISMPDeviceExtension**” class.

The asynchronous approach is easier to use than the synchronous since the application has only to call the function only once to send the whole chunk of data.

**Note:**

It is recommended to use the second method to exchange data since it is easier to implement.

### 4\_8\_3 ICTransaction Initialization

The initialization of an “**ICTransaction**” instance goes through the traditional initialization path:

- Create an auto-released instance by calling the shared initializer “**+(id)sharedChannel**” and retain it.
- Set the delegate that will receive stream events using the “**streamEventDelegate**” property or the “**forwardStreamEvents**” method. The delegate has to be a valid object implementing the “**NSSstreamDelegate**” protocol. The delegate is necessary only if the input and output streams would be managed asynchronously. The latter approach is always preferred to the synchronous stream management, and will be covered later in this section.
- Check the “**isAvailable**” property to ensure that the object was properly initialized and the communication session opened to the Companion.

After these three steps, the “**ICTransaction**” object is ready to send and receive data.

## 4\_8\_4 ICTransaction Disposal

To dispose the “**ICTransaction**” object and close the correspondent external accessory session opened to the Companion, the application has just to release the object.

## 4\_8\_5 State Changes

Like the other objects, “**ICTransaction**” inherits from “**ICISMPDevice**”, and as such, calls “**accessoryDidConnect**” and “**accessoryDidDisconnect**” callbacks to notify of its state changes over the time. The delegate should also compare the “**sender**” parameter to the “**ICTransaction**” object to make sure the latter emitted the events.

## 4\_8\_6 Sending/Receiving Data Using RAW Streams

The advantage of adopting an asynchronous approach to send and receive data using the streams provided by “**ICTransaction**” is that the stream events are scheduled in a different run-loop on a dedicated thread created by the iSMP library. Therefore, the developer won’t have to spawn other threads to manage read and write operations.

To handle stream events asynchronously, the developer will have to do a couple of things.

### 4\_8\_6\_1 Initialize Read and Write Buffers

The write buffer is a FIFO in which the data that will be sent to the Companion is appended. When a part of the data is delivered, it is removed from the buffer.

The read buffer is also a FIFO that is used to save the received data. When a part of the received data is processed, it is removed from the FIFO.

These objects can just be “**NSMutableData**” objects.

### 4\_8\_6\_2 Writing Data

Writing data to the Companion can be achieved in three steps:

1. Appending the new data at the end of the write FIFO. The access to the FIFO should be synchronize using a lock or @synchronized directive to avoid conflicts

#### Example:

```
[writeFifoLock lock];
[writeFifo appendData:newData];
[writeFifoLock unlock];
```

2. Calling the “**write:maxLength:**” method on the “**outStream**” property with NULL data to trigger a stream event and start sending the data asynchronously

#### Example:

```
[self.transactionObject.outStream write:NULL maxLength:0];
```

3. In the “**stream:handleEvent:**” callback, isolate the “**NSStreamEventHasSpaceAvailable**”, write as much data as possible by calling “**write:maxLength:**”, and update the write FIFO by removing what was sent.

#### Example:

```
-(void)stream: (NSStream*) aStream handleEvent: (NSStreamEvent) streamEvent {
```



```

Switch(streamEvent) {
case NSStreamEventHasSpaceAvailable:
if (aStream == self.transactionObject.outStream) {

    //Lock the access to the write Fifo
    [writeFifoLock lock];

    if([writeFifo length] > 0) {

        //Try to write the available data
        int bytes = [self.transactionObject.outStream write:[writeFifo bytes]
maxLength:[writeFifo length]];

        //Remove the written data from the Fifo
        if(bytes > 0) {
            [writeFifo setData:[writeFifo subdataWithRange:NSMakeRange(bytes, [writeFifo
length] - bytes)]];
        }
        [writeFifoLock unlock];
    }
    break;
}
}
}

```

## 4\_8\_6\_3 Reading Data

When some data is received by the iOS device and is available, “**stream:handleEvent:**” is called with a stream event type “**NSStreamEventHasBytesAvailable**”. The application can, then, read this data by calling “**read:maxLength:**” method on the “**ICTransaction**” object, and process it.

### Example:

```

-(void)stream: (NSStream*) aStream handleEvent: (NSStreamEvent) streamEvent {
    uint8_t inBuffer[1024];

    Switch(streamEvent) {
    case NSStreamEventHasBytesAvailable:

        if (aStream == self.transactionObject.inStream) {

            int read = [self.transactionObject read:inBuffer maxLength:sizeof(inBuffer)];

            if(read > 0) {

                //Append the data to the read Fifo
                [readFifo appendBytes:inBuffer length:read];

                //Process the data inside the Fifo
                [self processData];
            }
        }
        break;
    }
}

```



Among the events called by “ICISMPDeviceExtension”, only “didReceiveData:fromICISMPDevice:” is required and needs to be implemented. This callback is triggered when some data has been received on a raw channel and is ready to be processed by the application

## 4\_9 Using the barcode reader

“ICBarCodeReader” is the class that manages the barcode reader of the Companion (for models with embedded barcode readers). It provides an interface for configuration and control and provides the barcodes and the barcode reader events to applications through a set of callbacks declared within its associated “ICBarCodeReaderDelegate” protocol.



Companions come with two variants: one equipped with a barcode reader and one that is not. For the latter, the “ICBarCodeReader” class will not be able to connect to the barcode reader and the “isAvailable” property will always be equal to **NO**.

### 4\_9\_1 Initializing the barcode reader

The instantiation of an “ICBarCodeReader” object requires calling the shared constructor and setting the delegate that will be notified with the events:

- Creating and retaining an “ICBarCodeReader” object by calling “(id)sharedICBarCodeReader” class method. The instance returned by this method is auto-released, and needs to be retained.
- Setting the “delegate” property of the newly created instance. The delegate is a pointer to an object that should implement the “ICDeviceDelegate” and “ICBarCodeReaderDelegate” protocols. The delegate object will receive the events of “ICBarCodeReader”

At this stage, the “ICBarCodeReader” object is constructed, but the barcode reader is not started yet. A function call to “powerOn” must be performed in order to turn on the barcode reader and start using it. This function may take up to 1 second to execute and will return whether the barcode reader was successfully powered or not.



The barcode reader is not powered by default when the “ICBarCodeReader” object is constructed

### 4\_9\_2 Barcode reader configuration

After power on, the barcode reader is ready to be used and can be configured by the iOS application. In order to scan barcodes, the barcode reader must be configured with the barcode types that it should be able to read. These are also called symbologies and can be configured using the “enableSymbology:enabled:” or “enableSymbologies:symbologyCount:” methods.

The list of all supported symbologies is defined within the “eICBarCodeSymbologies” enumeration. This includes all the following barcode types:

iSMP Library Constant	Barcode Type
ICBarCode_EAN13	EAN 13
ICBarCode_EAN8	EAN 8

ICBarcode_UPCA	UPC-A
ICBarcode_UPCE	UPC-E
ICBarcode_UPCE1	UPC-E1
ICBarcode_EAN13_2	EAN 13 Add On 2
ICBarcode_EAN8_2	EAN 8 Add On 2
ICBarcode_UPCA_2	UPC-A Add On 2
ICBarcode_UPCE_2	UPC-E Add On 2
ICBarcode_EAN13_5	EAN 13 Add On 5
ICBarcode_EAN8_5	EAN 8 Add On 5
ICBarcode_UPCA_5	UPC-A Add On 5
ICBarcode_UPCE_5	UPC-E Add On 5
ICBarcode_Code39	Code39
ICBarcode_Interleaved2of5	Interleaved 2 of 5
ICBarcode_Standard2of5	Standard 2 of 5
ICBarcode_Matrix2of5	Matrix 2 of 5
ICBarcode_Codabar	Codabar
ICBarcode_AmesCode	AmesCode
ICBarcode_MSI	MSI
ICBarcode_Plessey	Plessey
ICBarcode_Code128	Code 128
ICBarcode_Code16k	Code 16k
ICBarcode_93	Code 93
ICBarcode_11	Code 11
ICBarcode_Telepen	Telepen
ICBarcode_Code49	Code 49
ICBarcode_Code39_Italian_CPI	Code 39 Italian CPI
ICBarcode_CodaBlockA	CodaBlock A
ICBarcode_CodaBlockF	CodaBlock F
ICBarcode_PDF417	PDF417
ICBarcode_GS1_128	GS1 128
ICBarcode_ISBT128	ISBT 128
ICBarcode_MicroPDF	MicroPDF
ICBarcode_GS1_DataBarOmni	GS1 DataBarOmni
ICBarcode_GS1_DataBarLimited	GS1 DataBar Limited
ICBarcode_GS1_DataBar_Expanded	GS1 Databar Expanded
ICBarcode_DataMatrix	DataMatrix
ICBarcode_QRCode	QR Code
ICBarcode_Maxicode	MaxiCode
ICBarcode_Aztec	Aztec



The barcode reader can be configured to read all or a subset of all available barcodes. To enable all barcode types, the **"ICBarcode\_AllSymbologies"** constant can be used with **"enableSymbology:enabled:"**. 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.**

#### 4\_9\_2\_1 Barcode reader default configuration

For each library substitution, an internal default configuration is applied at the first connection to the barcode reader.

The barcode reader's default configuration is given in the table below:

Parameter	Default Value
<b>Enable Turn off after good read (multi scan)</b>	Enabled
<b>Imager Mode</b>	1D & 2D
<b>Good Scan Beep</b>	YES
<b>Transmission of UPC-A as EAN13</b>	Disabled
<b>Transmission of UPC-E as UPC-A</b>	Disabled
<b>Transmission of EAN8 as EAN13</b>	Disabled



If the barcode reader is in an unknown state or if you want to force the default configuration you can reconfigure the barcode reader using the API `applyDefaultConfiguration`.

#### 4\_9\_2\_2 Barcode reader methods

The iOS app can use the methods provided by the “**ICBarcodeReader**” class in order to configure the barcode reader. Those are listed on the table below.

Method	Description
<b>(void)configureBarCodeReaderMode:(int)mode</b>	Switch from Single-Scan to Multi-Scan. In Single-Scan mode, the operator will have to press the barcode button for each item to scan
<b>(void)configureImagerMode:(int)mode</b>	Switch between 1D and 1D&2D mode
<b>(void)enableSymbologies:(int*)symbologies symbologyCount:(int)count</b>	Enable a set of barcode types / Disable all barcode types
<b>(void)enableSymbology:(int)type enabled:(bool)enabled</b>	Enable / Disable a barcode type
<b>(void)enableAimerFlashing:(BOOL)enabled</b>	Enable / Disable the flashing of the aimer
<b>(void)illuminationMode:(int)mode</b>	Set the illumination mode
<b>(void)lightingGoal:(int)goal</b>	Set the lighting goal
<b>(void)lightingMode:(int)mode</b>	Set the lighting mode
<b>(void)illuminationLevel:(int)level</b>	Set the illumination level
<b>(void)goodScanBeepEnable:(bool)enabled</b>	Enable / Disable good scan beep (hardware beep)
<b>(void)setBeep:(BOOL)enabled frequency:(int)frequency andLength:(int)length</b>	Change the hardware beep's tone
<b>(void)setScanTimeout:(int)timeout</b>	Set the barcode decoding timeout
<b>(BOOL)enableTrigger:(BOOL)enabled</b>	Enable / Disable the barcode reader's trigger button

<b>(void)enableTransmitUPCABarcodesAsEAN13:(BOOL)enabled</b>	Enable/Disable conversion of UPC-A barcodes to EAN-13 when decoded
<b>(void)enableTransmitUPCEBarcodesAsUPCA:(BOOL)enabled</b>	Enable/Disable conversion of UPC-E barcodes to UPC-A when decoded
<b>(void)enableTransmitEAN8BarcodesAsEAN13:(BOOL)enabled</b>	Enable/Disable conversion of EAN-8 barcodes to EAN-13 when decoded
<b>(void)applydefaultConfiguration</b>	Apply the default configuration to the barcode reader



The configuration set by the application is volatile and remains active while the barcode reader is not powered off. This means from the moment it is applied until the barcode reader is turned off or reset.

### 4\_9\_2\_3 Barcode reader retrieve configuration methods

The majority of the methods that help to configure the barcode reader have their counterparts to help get the active configuration. Those are listed in the following table:

Method	Description
<b>(int)getBarCodeReaderMode</b>	Get the barcode reader mode
<b>(BOOL)isSymbologyEnabled:(int)type</b>	Check if a barcode type is enabled
<b>(BOOL)aimerFlashing</b>	Get the aimer state
<b>(int)illuminationMode</b>	Get the illumination Mode
<b>(int)illuminationLevel</b>	Get the illumination level
<b>(int)lightingGoal</b>	Get the lighting goal
<b>(int)lightingMode</b>	Get the lighting mode
<b>(int)getScanTimeout</b>	Get the scan timeout
<b>(BOOL)isTriggerEnabled</b>	Get the trigger button's state

### 4\_9\_3 Reading barcodes

Barcode reader events are provided to applications through the callbacks listed in the table below. The “Calling Context” column indicates where those callbacks are called.

Callback	Description	Calling Context
<b>-(void)barcodeData:(id)data ofType:(int)type</b>	Barcode Scanned Event	Main Runloop
<b>-(void)triggerPulled</b>	Trigger Button Pushed Event	Main Runloop
<b>-(void)triggerReleased</b>	Trigger Button Released Event	Main Runloop
<b>-(void)unsuccessfullDecode</b>	Failed to Decode Barcode Event	Main Runloop
<b>-(void)configurationRequest</b>	Configuration Request Event	Main Runloop

The barcodes that are well decoded are provided to applications through the following callback:

**-(void)barcodeData:(id)data ofType:(int)type;**

The barcode is contained within the “**data**” object which is always of type “**NSString**”. The reason why an “**id**” pointer is used is because the barcode reader can be configured to take snapshots and return a bitmap. This has been, however, disabled because of the very low baud-rate on the iOS device’s serial link.

So, the application should call the “**isKindOfClass**” method on the “**data**” object, and check that it is of type “**NSString**”.

The “**type**” argument of the callback tells about the type of the scanned barcode. This is an “**eICBarcodeSymbolgies**” constant. The “**ICBarcodeReader**” class provides the following helper method to convert the barcode type constant into a string representation:

```
+(NSString *)symbologyToText:(int)type;
```

An alternate way to read barcode is the use of the functions **startScan** and **stopScan**. This method is not recommended as the decoding of the barcode will have to be performed programmatically and not left to the Companion.

When a scan session is started using **startScan**, it ends either when **stopScan** is called or after decoding a barcode, if the barcode reader is configured in Single Scan mode. The session may also stop if the barcode reader is turned off after calling **powerOff**.

#### 4\_9\_4 Releasing the barcode reader

The barcode reader can be turned off in three different ways:

- Releasing the “**ICBarcodeReader**” object
- Calling the “**powerOff**” method of “**ICBarcodeReader**”
- Losing the connection with the Companion, in which case the “**accessoryDidDisconnect**” callback is called on the delegate of the “**ICBarcodeReader**” object

The last point can happen as a result to different situations that involve:

- Application entering in background mode and then suspended by iOS.
- Companion rebooting
- iOS device unlinked



There is no need to call “**stopScan**” or “**powerOff**” when the delegate/application receives the “**accessoryDidDisconnect**” callback for the “**ICBarcodeReader**” object. The barcode reader and its LED are turned off automatically.

#### 4\_9\_5 Additional functions

Information about the Companion’s barcode reader (including its name, model and firmware version) can be retrieved using the following methods:

Method	Description
<b>(NSString *)scannerName</b>	Get the barcode reader name
<b>(NSString *)scannerModel;</b>	Get the barcode reader model
<b>(NSString *)scannerFirmwareVersion</b>	Get the barcode reader’s firmware version

This information can only be retrieved when the barcode reader is powered on. This means when the “**isAvailable**” property returns **YES**.

“**softReset**” function is used to reset the barcode reader. This is quicker way to reapply the default configuration then calling “**powerOff**” and “**powerOn**”. The volatile configuration of the barcode reader is lost and the “**configurationRequest**” callback is called as a result to this action. The application should then reconfigure the barcode reader as described in the previous paragraphs.

The “**playBeep**” request is used to inquire the barcode reader to play a customized sound with a certain frequency and duration. It is convenient in this case to turn off the good scan beep using the “**goodScanBeepEnable**” function.

## 4\_9\_6 Power Management Considerations

The barcode reader is the most consuming component of the Companion, and as such, iOS 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.
- **When the application enters the background mode, make sure to turn off the barcode reader**

## 4\_10 Using the Virtual SPP Channel

The virtual SPP channel is an additional feature that needs to be unlocked before being used. It allows you to have a Bluetooth SPP (Serial Port Profile) channel between the iOS device and an external Bluetooth device (not MFI certified) that is connected to the Companion. This communication is done using the class ICSP.

### 4\_10\_1 ICSP Initialization

The initialization of an “**ICTransaction**” follows few steps:

- Create an auto-released instance by calling the shared initializer “**+(id)sharedChannel**” and retain it.
- Set the “**delegate**” property to be notified with the events of “**ICSP**”. The delegate must conform to the “**ICISMPDeviceDelegate**” and “**ICISMPDeviceExtensionDelegate**” protocols. The latter has the required “**didReceiveData:fromICISMPDevice:**” that needs to be implemented in order to be able to receive data on the SPP channel.
- Check the “**isAvailable**” property to ensure that the object was properly initialized and the communication session opened.



It is very important to mention that the “**isAvailable**” property indicates the state of the transparent SPP channel created between the Companion and the iOS device is open and ready to route the SPP connection. It does not mean, however, that the third party Bluetooth device is connected. So far, there is no API that helps get the state of this device. The iOS application must, therefore, check it itself, by polling and listening on the link.



After these three steps, the “**ICSPP**” object is ready to send and receive data.

## 4\_10\_2 ICSPP Disposal

To dispose the “**ICSPP**” object and close the corresponding external accessory session opened to the Companion, the application has just to release the object.

It is important to note that once the “**ICSPP**” object is released, its “**delegate**” property is reset to **nil**, and needs therefore to be set if the application had to initialize the “**ICSPP**” object again.

## 4\_10\_3 State Change

Upon initialization, the SPP channel is opened by default. That's why it is convenient to check its “**isAvailable**” property. It is however necessary afterwards to keep track of the state changes of this communication channel, to know when and when it is not available.

The state of the SPP channel may change unpredictably (Companion reboot, Companion disconnected, ...). As a matter of fact, it is necessary for the delegate of “**ICSPP**” to implement the callbacks inherited from “**ICISMPDevice**” class:

- **accessoryDidConnect:(ICISMPDevice \*)sender;**
- **accessoryDidDisconnect:(ICISMPDevice \*)sender;**

Also, to make things clear, those callbacks are triggered whenever the external accessory session is broken or established, and not when the third party Bluetooth device is connected or disconnected.

## 4\_10\_4 Exchanging Data

In order to exchange data with a Bluetooth device connected to the Companion using the SPP profile, use the following methods and callbacks inherited from “**ICISMPDeviceExtension**”:

- Methods
  1. **SendData**
  2. **SendDataAsync**
  3. **SendString**
  4. **SendStringAsync**
- Callback
  1. **didReceiveData:fromICISMPDevice:**

## 4\_11 Using an external Bluetooth printer

This section describes how an iOS application can use a Bluetooth printer connected to a Companion.

### 4\_11\_1 Naming Conventions

All the methods of the “**IBP**” category of “**ICAdministration**” start with the “**iBP**” prefix. This makes it easy when initializing an “**ICAdministration**” object to point the methods that are used for Bluetooth printing. This rule applies also for other classes, structures or enumerations defined within the library, for instance the “**iBPBitmapContext**” or the “**iBPResult**” enumeration.

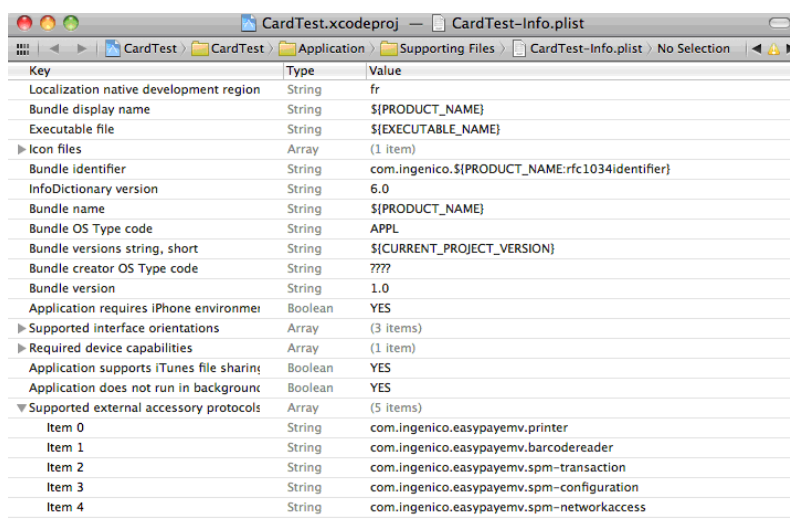
### 4\_11\_2 Application Requirements

An application built with the “**iSMP**” framework must declare the protocol names that are used for the communication with the Companion device. Those are mentioned in the chapter 4\_2\_2. Since the printing does only require the administration channel to be open, the iOS application has at least to add the following protocol name:

- **com.ingenico.easypayemv.spm-configuration**



This must be added to the file under the key “**Supported external accessory protocols**” as shown below:



Key	Type	Value
Localization native development region	String	fr
Bundle display name	String	\$(PRODUCT_NAME)
Executable file	String	\$(EXECUTABLE_NAME)
Icon files	Array	(1 item)
Bundle identifier	String	com.ingenico.{\$(PRODUCT_NAME:rfc1034identifier)}
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	\$(CURRENT_PROJECT_VERSION)
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Application requires iPhone environment	Boolean	YES
Supported interface orientations	Array	(3 items)
Required device capabilities	Array	(1 item)
Application supports iTunes file sharing	Boolean	YES
Application does not run in background	Boolean	YES
Supported external accessory protocols	Array	(5 items)
Item 0	String	com.ingenico.easypaymv.printer
Item 1	String	com.ingenico.easypaymv.barcode reader
Item 2	String	com.ingenico.easypaymv.spm-transaction
Item 3	String	com.ingenico.easypaymv.spm-configuration
Item 4	String	com.ingenico.easypaymv.spm-networkaccess

### 4\_11\_3 Bluetooth Printing API

The API provided by the “**iSMP**” framework consists of a set of methods that extend the “**ICAdministration**” class through an Objective-C category named “**iBP**”. So it is necessary to instantiate this class to be able to call the Bluetooth printing functions.

### 4\_11\_4 ICAdministration Initialization

The instantiation of an “**ICAdministration**” object is done exactly as described in [Erreur ! Source du renvoi introuvable.]. The application should get the shared instance of the class, retain it and set its “**delegate**” properties to be notified of events. The “**isAvailable**” property can be checked to determine whether the Companion is ready for communication.

### 4\_11\_5 Constants & Status Codes

The iSMP framework has a number of constants and error codes that the iOS developer should account for. All these constants and status codes are document in [Erreur ! Source du renvoi introuvable.].

The most important status codes that must be checked each time a printing method is used are the `iBPResult_<status>` codes and are defined as an `eiBPResult` enumeration.

In addition, the maximum size of bitmaps that can be printed using the “**iBP**” API can also be retrieved programmatically using the `iBPMaxBitmapWidth` and `iBPMaxBitmapHeight` read-only properties defined within the “**iBP**” category of “**ICAdministration**”

Property	Description
<code>-(NSInteger)iBPMaxBitmapWidth;</code>	Maximum Printed Bitmap Width – returns 384
<code>-(NSInteger)iBPMaxBitmapHeight;</code>	Maximum Printed Bitmap Height – returns 1024

### 4\_11\_6 API Functions

The methods that control the Bluetooth printer are listed in the table below.

Method	Type
<b>(iBPResult)iBPOpenPrinter</b>	Synchronous
<b>(iBPResult)iBPClosePrinter</b>	Synchronous
<b>(iBPResult)iBPGetPrinterStatus</b>	Synchronous
<b>(iBPResult)iBPPrintText:(NSString *)text</b>	Synchronous
<b>(iBPResult)iBPPrintBitmap:(UIImage *)image</b>	Synchronous
<b>(iBPResult)iBPPrintBitmap:(UIImage *)image lastBitmap:(BOOL)isLastBitmap</b>	Synchronous
<b>(iBPResult)iBPPrintBitmap:(UIImage *)image size:(CGSize)bitmapSize alignment:(UITextAlignment)alignment</b>	Synchronous
<b>(iBPResult)iBPPrintBitmap:(UIImage *)image size:(CGSize)bitmapSize alignment:(UITextAlignment)alignment lastBitmap:(BOOL)isLastBitmap</b>	Synchronous
<b>(iBPResult)iBPStoreLogoWithName:(NSString *)name andImage:(UIImage *)logo</b>	Synchronous
<b>(iBPResult)iBPPrintLogoWithName:(NSString *)name</b>	Synchronous

These methods will be explained in detail in the next paragraphs.

#### 4\_11\_6\_1 Open Printer

To open the printer, the application must call **“iBPOpenPrinter”**. This call will ask the Companion to connect to the Bluetooth printer with which it should be already paired.

Opening the printer is a synchronous operation that may take a variable time. It has a 15 seconds timeout but generally the method returns **“iBPResult\_OK”** just after 2 or 3 seconds, even if the printer is not connected (this is firmware specific). If the printer is turned on and paired with the device, the open operation should be successful and the printer ready to work and execute other requests. If the printer is somehow absent or not paired with the Companion, the application can determine its real status through the result of subsequent requests.

Note also, that it does not help to call **“iBPGetPrinterStatus”** after the call to open to check if the printer is ready or not. Indeed, the result of this function may be **“iBPResult\_OK”** for a certain amount of time until the terminal acknowledges the printer to be unavailable. So it is important to check the return code of each printing request, to have accurate information about the printer status.

#### 4\_11\_6\_2 Close Printer

The **“iBPClosePrinter”** closes the connection to the Bluetooth printer. It should be called when the printing is done, and not left open for a possible future print job. This function does not take too much time to execute (1 or 2 seconds). It returns **“iBPResult\_OK”** when successful. Get Printer Status

The **“iBPGetPrinterStatus”** requests the Bluetooth printer status at any given time and takes generally between 0.5 to 1 second to execute.

This function should be called by the application when a printing request fails, in order to know how to recover. For example, if the application tries to print text while no paper is left in the printer, the call to **“iBPPrintText”** will fail returning **“iBPResult\_KO”**. The application should then request the status of the printer to determine the nature of the error and try to recover.

There are many reasons, among the following, that may cause a printing to fail:

- Companion and iOS device not connected,
- No more paper in the printer,
- Paper container is left open,
- Printer is out of reach,
- Printer out of battery.

## 4\_11\_6\_3 Print Text

Text printing is performed using calls to **"iBPPrintText"**. This method takes as argument a text string object that must be composed of 512 characters at most and returns **"iBPResult\_OK"** if the request is successfully processed. The text is always printed at a new line and is automatically wrapped when the end of line is reached.

The text printed using this method can't be formatted. Its font, size and alignment are configured by default in the Companion software and can't be changed. If the iOS application needs to change these parameters, the library provides a helper class **"iBPBitmapContext"** that can be used to render text and images locally inside a bitmap context that can later be printed using the **iBPPrintBitmap** function. Note that this option increases the printing time.

### Hints:

- To insert a new line, **"iBPPrintText"** can be called with a NSString object containing a space character
- The new line character **"\n"** is interpreted as such when printing the text.

## 4\_11\_6\_4 Print Bitmap

The **"iBPPrintBitmap"** function prints a bitmap contained within a UIImage object. There are no restrictions on the bitmap's color configuration or format, providing that it is supported by iOS.

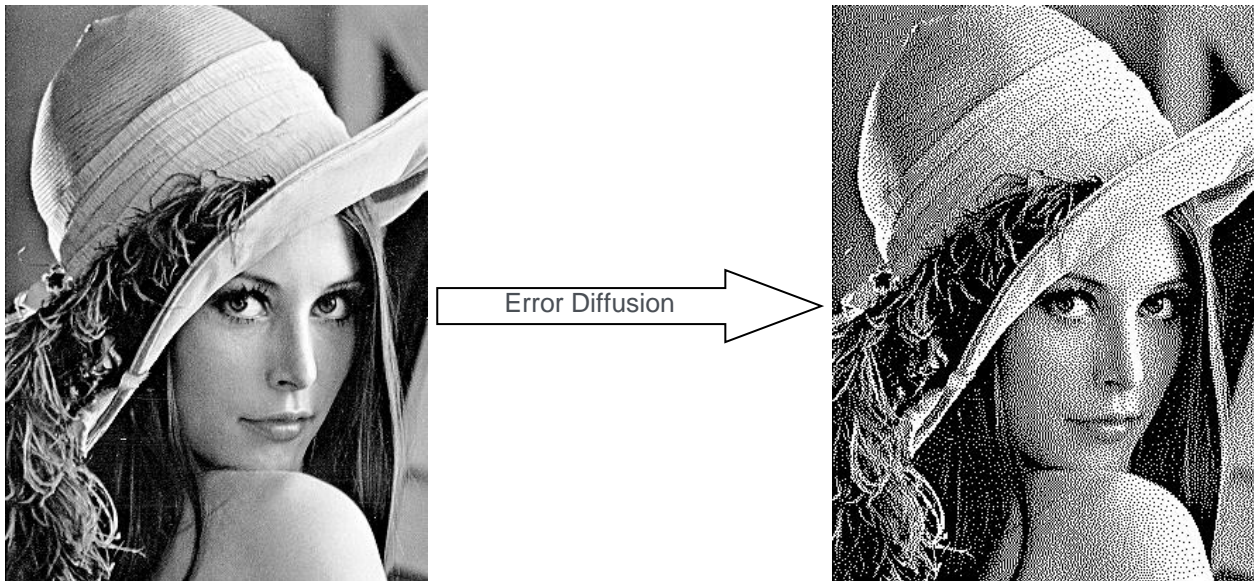
There are 2 cases where you must use the function **"iBPPrintLastBitmap"** instead of the **"iBPPrintBitmap"**:

- The bitmap you want to print more than 1024 in height
- You want to print many bitmaps consecutively without white space between each

Bitmaps undergo different transformations before being printed:

- Conversion to black and white since the Bluetooth printer does only support this type of bitmap. The conversion is based on the error diffusion algorithm.
- Scaling to make the bitmap fit within the maximum allowed dimensions (384 \* 1024). The scaling preserves the aspect ratio.

Example of a Bitmap before and after the conversion:



Important:

Final converted bitmaps are encoded so that each of their pixels is represented by one bit of data. Depending on the technical connectivity used between the iOS system and the Companion, the transfer time can be up to 8 seconds (when using the serial connector, with a baud rate of 57kb/s).

#### 4\_11\_6\_5 Print Bitmap, Size, Alignment

The “**iBPPrintBitmap:size:alignment**” methods gives, in addition to what “**iBPPrintBitmap**” does, the possibility to resize the printed bitmap and to align it within the printed receipt.

To resize the bitmap, the application should provide a “**CGSize**” structure that represents the size of the area in which the bitmap will be drawn. When the size of this area exceeds the maximum bitmap size (384 \* 1024 as mentioned previously), the maximum bitmap width and/or height values are used.

As with the “**iBPPrintBitmap**” function, use the corresponding “**iBPPrintBitmap>LastBitmap:size:alignment**” function in one of the following 2 cases:

- The bitmap you want to print is more than 1024 in height
- You want to print many bitmaps consecutively without white space between each

#### 4\_11\_6\_6 Store Logo

The “**iBPStoreLogo**” method stores a logo inside the Companion’s flash memory so that it can be reused lately without having to transfer it each time. The bitmap undergoes the exact same transformations as with the print bitmap API (conversion to black and white, and rescaling). It is stored inside the Companion until the latter is reset or shut down.

The “**iBPStoreLogo**” method takes as input a UIImage object that contains the bitmap to be stored, and the name of the file to which the bitmap will be saved inside the Companion. The name of the logo must be composed of 4 to 8 characters. If the name parameter has less of more than the specified number of characters, the call fails and returns “**iBResult\_WRONG\_LOGO\_NAME\_LENGTH**”.

Calling “**iBPStoreLogo**” method does only save the logo and does not print it. In order to be printed, the iOS application should explicitly call the “**iBPPrintLogo**” method with the adequate logo’s name.

Note:

The number of logos that can be stored is only limited by the size of the Companion's flash memory.

## 4\_11\_6\_7 Print Logo

The **"iBPPrintLogo"** method prints a logo previously loaded into the Companion using the **"iBPStoreLogo"** method. The method takes as argument the name of the logo. If the request is successful, the method returns **"iBPResult\_OK"**. If a logo with the provided name is not found, the method fails.

### Hint:

There is actually no API to check if a logo was already loaded. The iOS app can however try printing the logo with the name it is supposed to have. If this fails, it means that the logo does not exist or that the Companion has rebooted after the last time the logo was stored.

## 4\_12 Debugging your application

Tracing is very important when it comes to debug an application.

The **"ICISMDeviceDelegate"** protocol provides the following two callbacks used for trace logging:

Callback	Calling Context
<b>(void)logEntry:(NSString *)message withSeverity:(int)severity</b>	Random Runloop
<b>(void)logSerialData:(NSData *)data incoming:(BOOL)isIncoming</b>	Random Runloop

The first callback logs the majority of the library's internal function calls as well as many results and information that are necessary to analyze what is done behind the scenes. All these messages can be filtered based on the **"severity"** argument that gives the tracing level of the current message.

The tracing severity levels are defined within the **"SEVERITY\_LOG\_LEVELS"** enumeration and can be summarized with the following table:

Severity	Description
<b>SEV_DEBUG</b>	Debug Message
<b>SEV_INFO</b>	Information Message
<b>SEV_TRACE</b>	Trace Message
<b>SEV_WARN</b>	Warning Message
<b>SEV_ERROR</b>	Error Message
<b>SEV_FATAL</b>	Fatal Error Message
<b>SEV_UNKNOWN</b>	Undetermined Severity Message

### Example:

To only print the warning and error messages, the **"logEntry:withSeverity"** callback can be implemented as following:

```
- (void)logEntry: (NSString *)message withSeverity: (int)severity {
    if ((severity == SEV_WARN) || (severity == SEV_ERROR)) {
        NSLog(@"%@", message);
    }
}
```



The “**ICISMPDevice**” class provides also the following two helper methods to get a string representation of the severity levels:

**(NSString \*)severityLevelString:(int)level**  
**(const char \*)severityLevelStringA:(int)level**

The second callback “**logSerialData:isIncoming:**” logs the data exchanges between the iOS device and the Companion. The “**isIncoming**” parameter determines the way in which the data travels. The example below shows how this callback may be implemented.

```
(void)logSerialData:(NSData *)data isIncoming:(BOOL)isIncoming {
    NSLog(@"[Data:      %@] [Length:      %d] %@",      ((isIncoming)      ?      @"Companion→iDevice":
    @"iDevice→Companion", [data length], [data hexDump]);
}
```

The “**hexDump**” method is implemented inside the iSMP library within a category of NSData. This method returns an NSString object containing a hexadecimal representation of the bytes of an NSData object.

The category that implement “**hexDump**” is private and does not appear in any header file of the iSMP library. Therefore, when using it within the code, the following category declaration might be added to the project to avoid compiler warnings:

```
@interface NSData ()
-(NSString *)hexDump;
@end
```



**The tracing callbacks are called from an arbitrary thread on a random runloop.** So, their implementation code should not manipulate directly objects that are managed by the main thread. Think, in this case, about scheduling these calls on the main thread using helper methods like “**performSelector:onMainThread**”

In order to separate barcode reader traces from other classes’ traces, “**ICBarcodeReaderDelegate**” provides two callbacks that overload “**logEntry:withSeverity**” and “**logSerialData:isIncoming**” of “**ICISMPDeviceDelegate**” to keep only “**ICBarcodeReader**” traces and filter the others. Following are the two tracing callbacks:

Callback	Description	Calling Context
<b>(void)barcodeLogEntry:(NSString*)entry withSeverity:(int)severity</b>	Internal Logic Traces	Random Context
<b>(void)barcodeSerialData:(NSData*)data isIncoming:(BOOL)isIncoming;</b>	Exchanged Data Traces	Random Context

“**ICAdministration**” tracing callbacks are also separated from those of “**ICISMPDevice**” and are defined within the “**ICAdministrationDelegate**” protocol. These callbacks are optional which means that they may not be implemented.

Callback	Description	Calling Context
<b>(void)confLogEntry:(NSString*)message withSeverity:(int)severity</b>	Internal Logic Traces	Random Runloop
<b>(void)confSerialLogData:(NSData*)data incoming:(BOOL)isIncoming</b>	Exchanged Data Traces	Random Runloop



The callbacks are called from the thread on which the traces are logged, which is not necessarily the main thread. They also should return fast and not be blocked.

To get the traces thrown by “**ICTransaction**”, just implement the callbacks “**logEntry:withSeverity:**” and “**logSerialData:incoming:**” defined within the “**ICDeviceDelegate**” protocol as explained before.

To get traces of what does “**ICSPP**” behind the scenes and the data it exchanges on the serial link, the following callbacks inherited from “**ICISMPDevice**” can be used:

- logEntry:withSeverity:
- logSerialData:incoming:

## 4\_13 Sample code

The iSMP library’s release package contains few applications that implement some of the functionality of the “**ICAdministration**” class. These are:

Name	Version	Description
.\Samples\CompanionTestSample	1.05	Sample that demonstrates how to use your Companion control barcode reader, Bluetooth printer, interact with your Companion, ...
.\Samples\InteractivePayment	6.5	Sample showing how to exchange data with the terminal using the Transaction channel.

## 4\_14 Using the iOS simulator

XCode provides iOS devices simulators to help debugging an application without the need of the physical hardware.

Keep in mind that the simulator does not embed an embedded iAP chip and as such cannot be used to test the connection between your application and the Companion.

## 5 Signing the application

In order to be able to generate an iOS application (.ipa file) a signature certificate is requested. This certificate can only be obtained by applying to an iOS program. One of the following 2 programs can be used:

- iOS Developer Program. This program gives you access to the following features:
  - ability to debug the application on a device that has been added to the list of “development devices”
  - ability to install the final .ipa package on all the devices that are listed in the list of “development devices”
  - ability to submit the application to the AppStore
- iOS Enterprise Developer Program. This program gives you access to the following features:
  - ability to debug the application on a device that has been added to the list of “development devices”
  - ability to install the final .ipa package on all the devices that are listed in the list of “development devices”, providing that the ipa has been signed by the associated development certificate
  - ability to distribute the .ipa. According to Apple license agreement, the application can only be distributed internally in the company

In order to apply to the iOS Enterprise Developer Program, you will need a DUNS number. This number can be obtain at the D&B website (delay is 30 days).

Details about iOS developer programs can be found on Apple’s website.

If you have on-going or future iOS development project, make sure to apply to one of these program early enough to have Apple’s certificate on time for your internal testing, else you will have delay in your project.

The Apple registration process takes a few weeks so be sure to start it early enough.



---

**Ingenico R&D will no longer respond to any request related to certificate / test device management.**

---

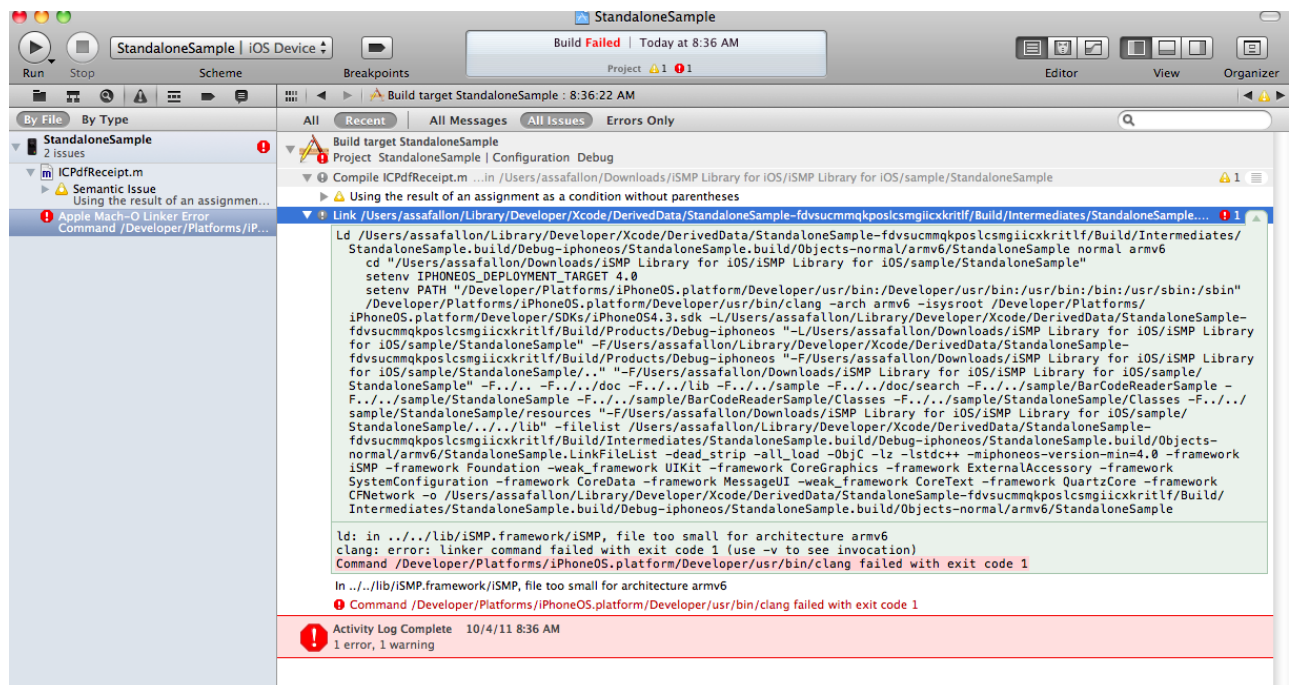


## 6 FAQ

This section highlights some issues that occurred during the development with the PCL library and applications around it, and the resolution.

### 6\_1 File Too Small when Compiling an App

This issue happens when the symbolic link named “iSMP” inside the iSMP framework is broken. This happens when moving the iSMP.framework folder from a file system that supports symbolic links to another one that does not (like the Windows NTFS file system). In such file system, the symbolic link file is transformed into a plain text file and completely loses its old attributes. This transformation is not reversible, which means that when the “iSMP” file is moved back to a Mac environment, it won’t regain its old state. As a result and since the “iSMP” symbolic link points to the binary file of the library, on the link process, the linker will link with a flat file and generates the following error.



There are two possible solutions:

1. Remove the broken “iSMP” symbolic link and recreate it using the following command on Mac:  
***ln -s ../libiSMP-x.y.a iSMP***  
Where “libiSMP-x.y.a” is the versioned binary of the iSMP library
2. Removed the broken “iSMP” and rename the “libiSMP-x.y.a” file to “iSMP” (the name of the framework).

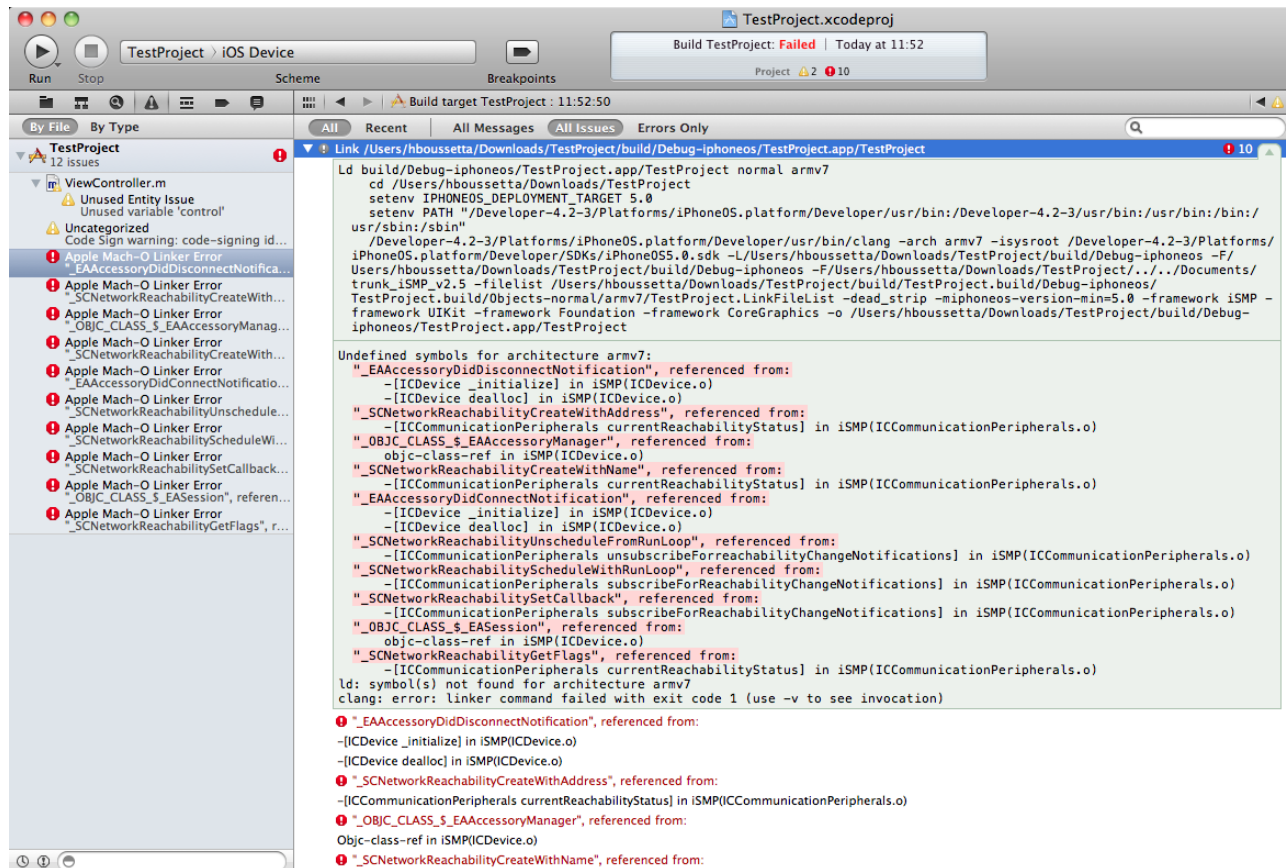
### 6\_2 Application not Compiling Properly – Missing Symbols

When the dependencies of the iSMP library are not referenced within the Xcode project, link errors will be encountered (figure below).

To fix them, make sure all the dependencies are added to the project. Those include:

- Foundation

- UIKit
  - CoreGraphics
  - SystemConfiguration
  - ExternalAccessory
- (Refer back to paragraph 3\_3 for more information)



## 6\_3 ICAministration Methods Unrecognized at Runtime

It may happen that an iOS application using the iSMP library crashes at runtime when calling methods of **ICTransaction** and says that those methods are unrecognized.

### Example:

```
<Error>: -[ICAdministration doTransaction:]: unrecognized selector sent to instance
<Error>: *** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: '-[ICAdministration doTransaction:]: unrecognized selector sent to instance'
```

This exception is thrown because the compiler was not provided the right options to load the code implemented by Objective-C categories. And as seen in the section 4\_6, the **ICAdministration** class has categories (like StandAlone for instance). Therefore, when linking an application with the iSMP library, it is mandatory to add the following compiler options as indicated in the section 4\_2\_2:

- "-all\_load"
- "-ObjC"

## 6\_4 Wrapping iSMP Library Objects with Singletons

This is a good practice that is highly recommended for several reasons:

- iSMP library objects are auto-released and need to be retained. The singleton will retain them and set itself as the direct delegate once and for all when it is initialized.
- The singleton is the direct receiver of the callbacks. It can provide a basic implementation and forward the callbacks to subsequent delegates, or just filter them (trace callbacks for example).
- The singleton can make the iSMP library object accessible as read-only to other objects by allowing them to call its services but not to change it.

Below is a code sample of how this can be.

```
@interface NetworkManager <ICDeviceDelegate, ICNetworkDelegate>

@property (nonatomic, readonly) ICNetwork network;

@end

-----

static NetworkManager * g_sharedNetworkManager = nil;

@implementation

@synthesize network = _network;

+(NetworkManager *)sharedNetworkManager {
    if (g_sharedNetworkManager == nil) {
        g_sharedNetworkManager = [[NetworkManager alloc] init];
    }
    return g_sharedNetworkManager;
}

-(id)init {
    if ((self = [super init])) {
        _network = [[ICNetwork sharedChannel] retain];
        _network.delegate = self;
    }
}

//implement callbacks
-(void)logEntry:(NSString *)entry withSeverity:(int)severity {
}

-(void)networkData:(NSData *)data incoming:(BOOL)isIncoming {
}

-(void)networkWillConnectToHost:(NSString *)host onPort:(NSUInteger)port {
}

...

@end
```

## 6\_5 Double Definition for Class ICDevice

As mention in paragraph 3\_4 , the “**ICDevice**” class is already defined in one of Apple’s private frameworks, which is the “**ImageCapture**” framework. When loading both frameworks, the following warning is thrown on the iOS traces:

```
Class ICDevice is implemented in both “iSMP.framework” and  
“/System/Library/PrivateFrameworks/ImageCapture.framework/ImageCapture”.  
One of the two will be used, which one is undefined.
```

This conflict seems to make the application linked with the iSMP library crash when calling one of the methods of “**ICDevice**”. There is no solution right now but to rename the “**ICDevice**” class to something else.

As a consequence, “**ICDevice**” was renamed into “**ICISMPDevice**” since iSMP library version 3.2.

This Document is Copyright © 2014 by INGENICO Group. INGENICO retains full copyright ownership, rights and protection in all material contained in this document. The recipient can receive this document on the condition that he will keep the document confidential and will not use its contents in any form or by any means, except as agreed beforehand, without the prior written permission of INGENICO. Moreover, nobody is authorized to place this document at the disposal of any third party without the prior written permission of INGENICO. If such permission is granted, it will be subject to the condition that the recipient ensures that any other recipient of this document, or information contained therein, is held responsible to INGENICO for the confidentiality of that information.

Care has been taken to ensure that the content of this document is as accurate as possible. INGENICO however declines any responsibility for inaccurate, incomplete or outdated information. The contents of this document may change from time to time without prior notice, and do not create, specify, modify or replace any new or prior contractual obligations agreed upon in writing between INGENICO and the user.

INGENICO is not responsible for any use of this software, which would be non-consistent with the present document.

The *Bluetooth*® word mark and logos are registered trademarks owned by Bluetooth SIG, Inc. and any use of such marks by INGENICO is under license.

Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

All trademarks used in this document remain the property of their rightful owners.”