Step-by-Step Guide for Integrating NConnect Framework for iOS

24 July 2023
Version: 3.1

Contents

Introduction

Welcome to the NConnectFramework integration guide for iOS, macOS, and iPadOS! The NConnectFramework is a powerful Swift-based framework that facilitates seamless communication with RFID (Radio-Frequency Identification) reader devices. This document aims to provide developers with a comprehensive understanding of the framework's components and their respective roles, allowing for a smooth and efficient integration process.

Understanding the NConnectFlow

1. **Obtaining the Reader Instance**: The integration begins with obtaining an instance of the rfid reader from the RfidFactory. This interface serves as the entry point for interacting with RFID reader devices.

2. **Configuration of Scanning Parameters**: Once the reader interface is acquired, it is essential to configure it with appropriate values for various scanning parameters. These parameters define the behavior of the RFID reader during scanning operations.

3. **Registering an RfidEventListener**: To receive real-time updates and notifications from the RFID reader, an RfidEventListener is registered with the RfidReader. This listener acts as a callback mechanism, enabling the implementing application to respond to events raised by the RFID reader.

4. **Initiating Scanning**: After the necessary configurations and event listener registration, the RfidReader is instructed to commence scanning for RFID tags or other operations.

5. **Asynchronous Callbacks**: As the RFID reader detects and processes tags or other events, the relevant data is asynchronously relayed to the implementing application. This is achieved through callbacks to the RfidEventListener registered with the RfidReader. These callbacks provide access to real-time data, allowing your application to perform desired actions in response to the RFID reader's activities.

Let's dive into the integration journey and unlock the possibilities that the NConnectFramework brings to your application!

System Requirements

Supported Platforms

iOS
iPadOS
macOS

Minimum OS version required:

iOS: 13.0
iPadOS: 13.0
macOS: 13.0

Tools required:

XCode version 14.3.1 or later

Installation

Drag and drop the framework files (usually with a .framework extension) into your Xcode project.

Make sure to select the appropriate target(s) when prompted to add the files.

After installation, ensure that the NConnectFramework is listed in your project's "Frameworks and Libraries" section in Xcode.

If you're using Cocoapods, check that the `import NConnectFramework` statement works correctly in your Swift files.

Integration Steps

Before we begin make sure app has following permission
`Bluetooth`
`Location`

Getting RFID Reader Instance

In order to establish communication with the RFID reader device, you need to obtain an instance of the RFID reader. The NConnectFramework provides a convenient method to achieve this. Here's how you can do it:

```
import NConnectFramework

do {
   // Specify the readerMake, hostname, and license parameters as per your
reader's configuration.
   let reader = try RfidFactory.getRfidReader(make: readerMake, hostname:
hostname, license: license)
   // 'reader' now holds the instance of the RFID reader.
} catch let error {
   // Handle any errors that might occur during the reader initialization.
   print("Error occurred while obtaining the RFID reader instance:
\(error.localizedDescription)")
}
```

Explanation:

The `RfidFactory.getRfidReader(make:hostname:license:)` method is used to create an instance of the RFID reader.

You need to provide the appropriate values for the `readerMake`, `hostname`, and `license` parameters when calling this method. These values will vary based on the type and model of your RFID reader device, and the licensing requirements of the NConnectFramework.

Handling Events and Registering Event Listener

In this step, we'll register an event listener that implements the `RfidEventListener` protocol to handle various events raised by the RFID reader, such as tag data, errors, connection status changes, and scan triggers. Here's how you can do it:

```
import NConnectFramework

// Create and register the event listener to handle RFID reader events.
let eventListener = createEventListener()

func createEventListener() -> RfidEventListener {
   class RFIDEventListenerImplementation: RfidEventListener {
       func handleData(tagData: String) {
           // Handle tag data received from the RFID reader.
           var response = [String: Any?]()
           response["event"] = "handleData"
           response["readerMac"] = mac
          // response["readTag"] = DataConvertor.hexToAscii(hexString:
tagData)
           response["readTag"] = tagData
           // Code for further processing of tag data...
       }

       func handleError(errorCode: NConnectFramework.ErrorCode,
description: String) {
           // Handle errors raised by the RFID reader.
           var response = [String: Any?]()
           response["event"] = "handleError"
           response["readerMac"] = mac
           response["error"] = errorCode
           // Code for error handling...
       }

       func handleEvent(eventCode: NConnectFramework.EventCode,
description: String) {
           // Handle various events raised by the RFID reader.
           var eventCodeStr: String
           switch eventCode {
```

```
           case .CONNECTED:
               eventCodeStr = "CONNECTED"
           case .DISCONNECTED:
               eventCodeStr = "DISCONNECTED"
           case .TRIGGER_ON:
               eventCodeStr = "TRIGGER_ON"
           case .TRIGGER_OFF:
               eventCodeStr = "TRIGGER_OFF"
           default:
               eventCodeStr = ""
           }
           // Code for further processing of events...
       }
   }
   return RFIDEventListenerImplementation()
}

// Register the event listener with the RFID reader.
reader.registerListener(listener: eventListener)
```

Explanation:
The `RfidEventListener` protocol provides three required methods: `handleData`, `handleError`, and `handleEvent`. These methods are called when the corresponding events occur.

In this example, we've implemented a custom class `RFIDEventListenerImplementation` that conforms to the `RfidEventListener` protocol. It handles tag data, errors, and various event codes raised by the RFID reader.

The `handleData` method is called when tag data is received from the RFID reader.

The `handleError` method is called when an error occurs during communication with the RFID reader.

The `handleEvent` method is called for events like CONNECTED, DISCONNECTED, TRIGGER_ON, TRIGGER_OFF, and any other events that may occur in the future.

The `createEventListener()` function creates an instance of the `RFIDEventListenerImplementation` class and returns it as an `RfidEventListener`.

RFID Reader Connection

Establishing a connection to the RFID reader is a crucial step in the integration process. Once you have obtained the RFID reader instance, you can attempt to connect to the reader using the `connect()` method. Upon successful connection, the RFIDEventListener will receive the `CONNECTED` event, indicating that the RFID reader is now connected and ready for further operations.

```
// Assuming 'reader' is the RFID reader instance obtained previously.

do {
   // Attempt to connect to the RFID reader.
   try reader.connect()
} catch let error {
   print("Error occurred while connecting to the RFID reader:
\(error.localizedDescription)")
}
```

Explanation:
The `connect()` method is called on the RFID reader instance to initiate the connection with the RFID reader specified during the creation of the reader instance using `RfidFactory.getRfidReader`.

If the connection to the RFID reader is successful, the `RFIDEventListener` registered with the reader will receive the `CONNECTED` event in the handleEvent method.

RFID Reader Operations

In this step, we will cover some essential RFID reader operations, such as starting and stopping scanning, setting power levels, and retrieving battery information.

Starting and Stopping Scanning

Scanning is the process of detecting RFID tags in the reader's range. You can start and stop scanning using the `startScan()` and `stopScan()` functions, respectively.

```
import NConnectFramework

// Assuming 'reader' is the RFID reader instance obtained previously.
// Start scanning for RFID tags.
reader.startScan()

// Stop scanning for RFID tags.
reader.stopScan()
```

Setting Power Level

The power level determines the strength of the RFID reader's signal, which affects the reading range and sensitivity to tag detection. You can set the power level using the `setPower(power:)` function. The power parameter should be a Double value between 0.0 and 30.0, where 0.0 represents the lowest power level, and 30.0 represents the highest.

```
import NConnectFramework

// Assuming 'reader' is the RFID reader instance obtained previously.

let powerLevel: Double = 20.0 // Replace with your desired power level
value.

do {
   try reader.setPower(power: powerLevel)
   // Power level set successfully.
} catch let error {
   print("Error occurred while setting power level:
\(error.localizedDescription)")
}
```

Retrieving Power Level

You can retrieve the current power level set on the RFID reader using the `getPower()` function.

```
import NConnectFramework

// Assuming 'reader' is the RFID reader instance obtained previously.

```

```
do {
   let currentPowerLevel = try reader.getPower()
   print("Current Power Level: \(currentPowerLevel)")
} catch let error {
   print("Error occurred while retrieving power level:
\(error.localizedDescription)")
}
```

Retrieving Battery Level

To check the battery level of the RFID reader, you can use the `getBatteryLevel()` function. The function returns a Double value representing the battery level, where 0.0 indicates a fully discharged battery, and 100.0 indicates a fully charged battery.

```
import NConnectFramework

// Assuming 'reader' is the RFID reader instance obtained previously.

do {
   let batteryLevel = try reader.getBatteryLevel()
   print("Battery Level: \(batteryLevel)%")
} catch let error {
   print("Error occurred while retrieving battery level:
\(error.localizedDescription)")
}
```

Best Practices

Order of Operations

Ensure you register the `RFIDEventListener` with the reader before attempting to connect to the RFID reader using `connect()`.

Following the correct order of operations ensures that you can receive the `CONNECTED` event if the connection is successful and be ready to handle further RFID reader events.

Avoid Method Conflicts

Avoid calling any RFID reader method (e.g., `startScan()`, `stopScan()`, `setPower(power:)`, etc.) while scanning is already in progress.

Conflicting method calls during scanning can lead to unexpected behavior or errors in RFID reader operations.

Implement a mechanism to track the scanning state and ensure that only one scanning operation is performed at a time.

Proper Error Handling

Always handle errors that may occur during RFID reader operations, such as connection errors, power level setting errors, or tag data handling errors.

Proper error handling ensures that your application gracefully recovers from any unexpected situations and provides appropriate feedback to users.

Event Handling

Handle all RFID reader events appropriately and respond to them accordingly.

Events like `CONNECTED`, `DISCONNECTED`, `TRIGGER_ON`, `TRIGGER_OFF`, and others provide valuable information about the RFID reader's state and activities.

Implement event handlers that perform the necessary actions based on the event received.