Contents

Infineon NFC Tag Side Controller (Smack) SDK for iOS

This repository contains the Infineon NFC Tag Side Controller (Smack) SDK for iOS.

> ***Note on energy harvesting:*** As the client app communicates with the NAC1080 hardware, some of the transmitted energy is harvested in a capacitor. This enables the usage of this energy for other purposes like moving a motor. However the capacitor can be discharged very quickly. If more energy is needed for the desired action to complete, you must keep supplying more energy. To achieve this, just keep communicating with the hardware after your energy consuming action has started, e.g. by reading a progress status.

Features

The SmackSDK is split up into multiple APIs in order to provide an easy and self-contained access for every use case.

- Lock Communication: High-level API for controlling smart locks
- Mailbox Communication: Low-level API for communicating directly with the Mailbox of a SmAcK hardware
- Motor Control: Private API for modifying the behavior of the motor
- Measurement: API that provides access to request certain measurement data from a device

Initialization

Every API can be used stand-alone by passing a target and optionally a configuration. When communicating with real hardware, use the SmackTarget.device(), otherwise a simulation can be requested via SmackTarget.emulation.

It is strongly adviced to only use one instance of every API. Example for an initialization:

```
let config = SmackConfig(
                 logging: CombinedLogger(
                     debugPrinter: DebugPrinter(),
                     logPrinter: CustomPrinter
                 )
             )
let client = SmackClient(config: config)
let target = SmackTarget.device(client: client)
let mailboxApi = MailboxApi(target, config: config)
let lockApi = LockApi(target, config: config)
```

Logging

The Logger interface is used for logging any communication between the SDK and the SmAcK Hardware.

When initializing a SmackClient you can also pass your own implementation as a `CombinedLogger`. This collection contains two loggers that will be used for the communication. The debugPrinter is a `Logger` that is used for logging in the debug console while the logPrinter is used for getting back all important logs to the client. For the last case you need to declare an `Logger` implementation in the app and pass it to the `CombinedLogger`.

Attention: This should not be used in production because sensitive information like the lock key can show up in logs.

Log Levels

- verbose: For additional information around the above levels
- debug: For low level information about sent and received byte arrays
- info: For high level calls like `writeDataPoint()` / `readDataPoint()`
- warning: For critical messages
- error: For exceptions and errors

Deployment

Deploy SDK as framework:

- update the framework version and the versions of its dependencies in the SmackSDK.podspec file
- merge into master
- the new framework will be automatically uploaded to Confluence and merged into the develop branches of the Sample and Showcase apps

Usage

Integrate SmackSDK.framework into client using Cocoapods:

- create a folder for the SmackSDK framework in your project root (e.g. `SmackSDK`)
- extract the zipped framework into the new framework folder
- open client project in Xcode
- add `pod 'SmackSDK', :path => './SmackSDK'` to your Podfile
- add the following post install hook to your Podfile so the SmackSDK.framework can see its dependencies at runtime

```
post_install do | installer |
  installer.pods_project.targets.each do |target|
      target.build_configurations.each do |config|
```

```
            config.build_settings['BUILD_LIBRARY_FOR_DISTRIBUTION'] = 'YES'
        end
    end
end
```

- run `pod install` in your project root
- import SmackSDK where needed and use its features

Apps

Currently the following apps use the SmackSDK:

- Smack Sample: Example for implementing and using the SmackSDK
- Smack Showcase: Showcase for a SmAcK-enabled smart lock

License

Mailbox

The mailbox acts as cache for the communication between client and firmware. Its address and size are dynamic and can be read, while its content can be read and write from.

Functions

Following functions are available:

MailboxApi	Parameters	Response	Description
observeChargeLevel	LockKey	ChargeLevel	Requests the current charge level that will be returned as a stream
test	-	MailboxTestResult (includes bytes sent, received and divergent)	Test the mailbox connection by writing random messages
resetData	-	Void	Resets the history of a lock and the encryption key
readWord	index	word	Read one word from the mailbox at the given index
writeWord	index, word	[Byte]	Write one word to the mailbox at the given index
readDataPoint	DataPoint, data	[Byte]	Read data with a given length from a data point
readDataPointWithDelay	DataPoint, data, delay	[Byte]	Read data with a given length from a data point after a given delay
writeDataPoint	DataPoint, data	[Byte]	Write data to a data point

MailboxApi	Parameters	Response	Description
callAppFunction	index, data	The first 64 bytes of the mailbox content	Calls one of the 16 app functions with the given index and writes the data to the mailbox

NAC1080 Firmware Compatibility

The SmAcK NAC1080 has the ability to read and write mailbox words implemented in ROM Lib and IFX NVM Lib. Thus reading and writing words can be done without any custom firmware.

For any further functionality, like handling data points, you need to create and flash a firmware made with the SmAcK Firmware SDK.

Data Points

Data points persist data outside of the mailbox within the firmware. They can be read or written via the mailbox which acts as a cache for the data from a data point. Communication with a data point consists of four fixed and n dynamic requests depending on the data to read or write: .dataPointCallAppFunction, .dataPointHeader and .dataPoint, then come the .data requests which are split up into chunks per word and lastly .validate which advises the firmware to calculate its state according to the parameters set before. Validating a data point takes two steps: first reading the header word and checking bitwise whether its flag are a valid response, second reading the data point word and comparing it with the data that was being written before.

Requests

Following requests can be send to the mailbox:

MailboxRequest	Parameters	Response	Description
validate	-	return code	Advises the firmware to calculate its state according to the parameters set within the mailbox and return a return code as result
readAddress	-	start index	Retrieve the current address of the mailbox
readWord	index	word	Read one word from the mailbox at the given index
writeWord	index, word	-	Write one word to the mailbox at the given index
dataPointCallAppFunction	index	-	Prepares a data point request at the given request
dataPointHeader	words	-	Specifies meta data for the data point request
dataPoint	data point, action, words	-	Declares a data point that is going to be read from or written to

MailboxRequest	Parameters	Response	Description
data	index, words	word (if reading)	Reads or writes data at the given index to the data point declared before

Error handling

Following errors may be returned from this component:

MailboxError	Reason
wordEmpty	Word is empty but must not be
indexOutOfRange	Mailbox address is out of bounds
invalid	Invalid with a given return code
headerResponseValidationFailed	Data point header has an invalid state
dataPointHeaderResponseValidationFailed	Data point header response has an invalid state
logStatusUnknown	The status of a single log entry is unknown
logUserNameEmpty	The user name of a single log entry is empty
wrongKey	The passed key is wrong
callAppFunctionValidationFailed	Validation for callAppFunction failed

Examples

Initialize the mailbox

To communicate with the mailbox, you first need a MailboxApi with a client. The SmackClient can be used to communicate with supported NFC hardware.

```
class MainViewModel : ViewModel() {

    private var mailboxApi: MailboxApi!
    private var logger: Logger = DebugPrinter()

    public init() {
        let config = SmackConfig(logging: CombinedLogger(debugPrinter: logger), delegate: self)
        let client = SmackClient(config: config)
        let target = SmackTarget.device(client: client)
        self.mailboxApi = MailboxApi(target: target, config: config)
    }
}
```

Reading a data point

Assuming you already initialized the mailbox as described above, the mailbox function readDataPoint() can be called to read a data point.

```
mailboxApi.readDataPoint(dataPoint: MailboxDataPoint.chargePercent(key: dummyKey)) { result in
    switch result {
    case .success: ...
    case .failure(let error): ...
    }
}
```

Writing a data point

Assuming you already initialized the mailbox as described above, the mailbox function writeDataPoint() can be called to write to a data point.

```
mailboxApi.writeDataPoint(
    dataPoint: MailboxDataPoint.chargePercent(key: LockKeydummyKey),
    data: [1, 2, 3]
) { result in
    switch result {
    case .success: ...
    case .failure(let error): ...
    }
}
```

Note: The length of the provided data must match the data type of the given data point.

Calling an app function

```
mailboxApi.callAppFunction(index: 5, data: [1, 2, 3]) { result in
    switch result {
    case .success(let byteArrays): ...
    case .failure(let error): ...
    }
}
```

Smack Communication

Smack Protocol

The Smack protocol is a proprietary protocol from Infineon.