Contents

Infineon NFC Tag Side Controller (Smack) SDK for Android

> ***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

- Lock Communication
- Mailbox Communication
- Motor Control
- Measurement

A Note on StateFlow vs LiveData

In the examples all SDK-Flows are shared with the UI through a `StateFlow`. Please note that you could convert the `Lock-Flow` to a `LiveData` as well. But then you must make sure that the `Flow` is only collected by one `LiveData` at a time. *Caution:* `LiveData` has a 5 seconds delay in which it will still collect the `Flow`, even if there are no observers registered anymore. This can cause `java.io.IOException: Only one TagTechnology can be connected at a time` in the SmAcK SDK when navigating between two `Fragments` which use two different `LiveData` instances. It's generally recommended to use this SDK with Flows as they give you more control over what data is cached or emitted.

To collect a `StateFlow` in your Fragment you should use `repeatOnLifecycle` which was introduced in the 2.4.0 version of lifecycle-runtime-ktx library

```
viewLifecycleOwner.lifecycleScope.launch {
    viewLifecycleOwner.lifecycle.repeatOnLifecycle(Lifecycle.State.STARTED) {
        lockActionViewModel.viewState.collect { binding?.setState(it) }
    }
}
```

Usage

To start using this SDK in your project, copying the sdk's .aar file into the `libs` folder of your project and adjust the `dependencyResolutionManagement`

in your `settings.gradle` file to recognize the libs folder:

```
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()
        flatDir {
            dirs 'libs'
        }
    }
}
```

Then add the SDK dependency to the `dependencies` in the `build.gradle` file of your app module:

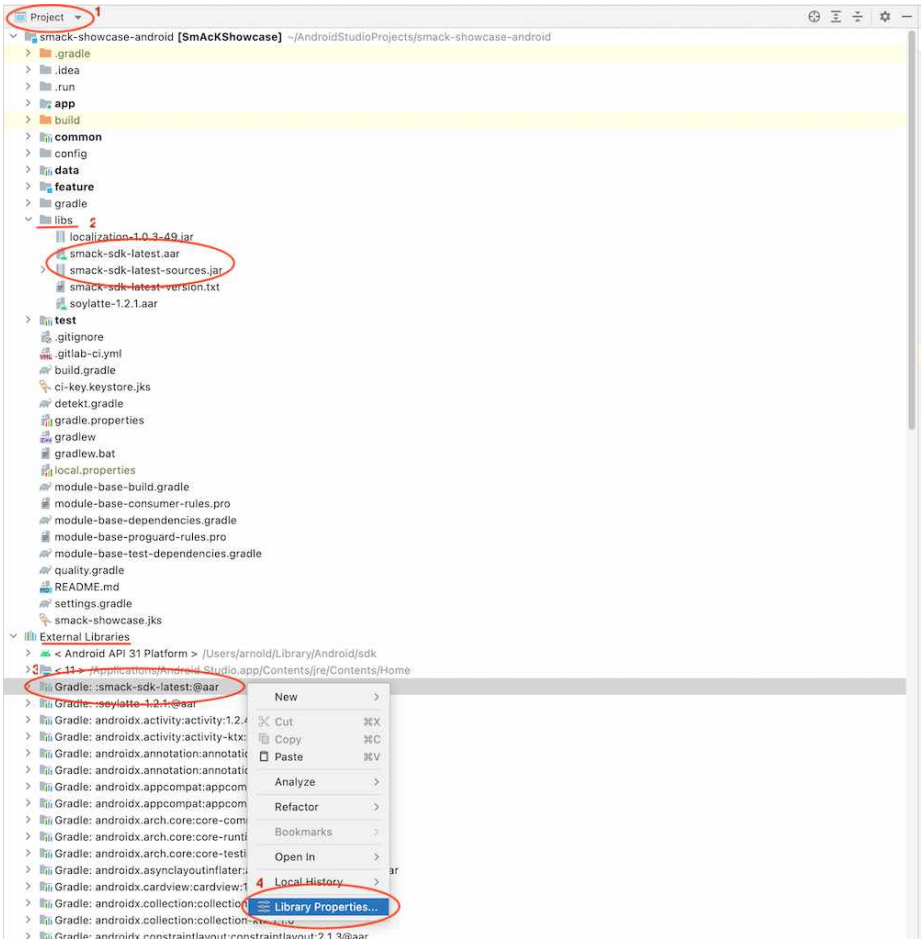
```
    implementation(name:'smack-sdk-android-<version>', ext:'aar')
```

Replace with the version of your .aar file in the upper example.

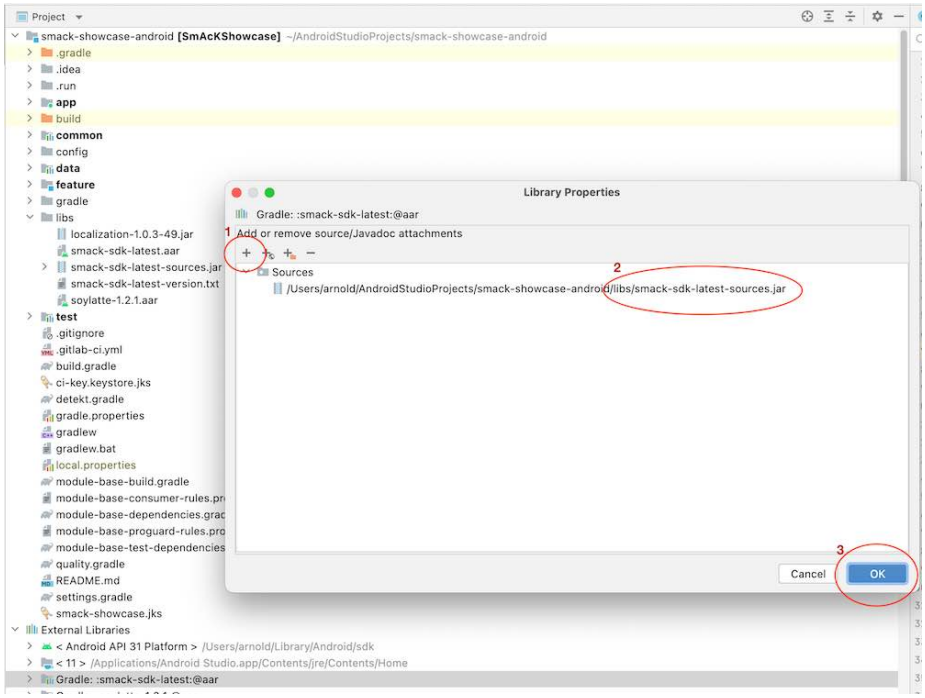
Add public sources and javaDoc comments to Android Studio

When defining a local gradle dependency like described above, the sources and javadoc comments are not visible automatically in Android Studio.

To add them you need to:



- Select the *Project* View in the left pane
- Make sure that you have a libs folder with `.aar` and `-sources.jar` inside
- Open *External Libraries* and search for `Gradle: :smack-sdk-latest:@aar`
- Right-click the library and choose *Library Properties*



- Click on Plus and add the `-sources.jar` path to the sources list
- Click OK to save

```kotlin
package com.infineon.smack.sdk.application.lock

import ...

/**
 * The [LockApi] is the main entry point for communication with SmAcK NFC locks.
 *
 * [SmackLockApi] is an example implementation of lock functions based on a specification
 * defined by Infineon, i.e. work flow, datapoint definition, key management.
 * It works only with a lock design which has the corresponding showcase firmware installed in NAC1080!
 *
 * If you implemented your own Smart Lock Firmware, you can provide your own implementation of LockApi.
 *
 * **How-To-Use:**
 *
 * Observe the lock [Flow] and map it to your ViewState as needed.
 *
 * Before you can perform any actions on the [Lock], you need to authenticate by calling [LockApi.initializeSession].
 *
 * Then you can use LockApi with the [Lock] to read lock information or lock and unlock the Lock
 * ([getChargeLevel], [lock], [unlock], [getHistory])
 *
 * @see [com.infineon.smack.sdk.application.lock.SmackLockApi]
 */
@Suppress( ...names: "TooManyFunctions")
interface LockApi {

    /** The [Flow] containing the [Lock] for use with this LockApi */
    fun getLock(): Flow<Lock?>

    /**
     * Registers a new lock key on a brand new lock.
     *
     * @param lock the [Lock] where to set the [LockKey]
     * @param userName Name of the Lock-User. May be up to 40 bytes long (after decoding with UTF-8)
     * @param dateTimeInSeconds The current date and time in seconds since 01.01.1970 (Epoch)
     * @param supervisorKey the ASCII encoded supervisor key must contain exactly 16 characters
     * (included in the locks original packaging)
     * @param password the ASCII encoded lock password to use (maximum length: 16 characters)
     *
     * @throws IllegalArgumentException if the user name is too long
     * @throws IllegalArgumentException if the password or the supervisor key is too long
     * @throws WrongKeyException when the firmware does not know the provided supervisor key
     * @throws LockIdValidationException if the lock id validation fails
     */
    suspend fun setLockKey(
        lock: Lock,
        userName: String,
        dateTimeInSeconds: Long,
        supervisorKey: String,
        password: String
    ): LockKey
```

- Now you can open public SDK classes and view their documentation

Initialization

Three steps are needed to use the `SmackSdk` to handle nfc events in your application, described in this chapter.

1. Register for NFC events (`Intents`)
2. Initialize the `SmackSdk`
3. Forward NCF `Intents` to the `SmackSdk`

Register for NFC events

To receive NFC intents in your activity, an intent filter must be added to the `<activity>` tag in the AndroidManifest.xml file.

`<intent-filter>`

```
    <action android:name="android.nfc.action.TAG_DISCOVERED" />
    <category android:name="android.intent.category.DEFAULT" />
    <data
        android:host="ext"
        android:pathPrefix="/${applicationId}:device"
        android:scheme="vnd.android.nfc" />
</intent-filter>
```

The app can be started by scanning an NFC tag containing an Android Application Record with your apps package name.

Initialize the SmackSdk

In order to communicate with SmAcK hardware you need to initialize a `SmackSdk` with the `Activity` which will receive the NFC tag intents. Register the SDKs `SmackLifecycleObserver` in `onCreate()` by calling the SDKs `onCreate(activity: Activity)` function.

Use `SmackSdk.Builder` to initialize the `SmackSdk`. All setter functions are optional. The initialized `SmackSdk` object provides a `SmackLifecycleObserver`, a `SmaclClient` and all three API objects (`SmackApi`, `MailboxApi` and `LockApi`) for further use in your implementation.

```
class MainActivity : AppCompatActivity() {

    private val viewModel: MainViewModel by viewModels()
    private lateinit var binding: MainActivityBinding

    // Initialize optional classes via dependency injection
    // or manually in init {} or onCreate().
    lateinit var smackClient: SmackClient
    lateinit var nfcAdapterWrapper: NfcAdapterWrapper
    lateinit var coroutineDispatcher: CoroutineDispatcher

    private val smackSdk: SmackSdk by lazy {
        SmackSdk.Builder(this)
            // All setters are optional!
            // SmackClient defaults to DefaultSmackClient(NoOpSmackLogger())
            .setSmackClient(smackClient)
            // NfcAdapterWrapper defaults to AndroidNfcAdapterWrapper()
            .setNfcAdapterWrapper(nfcAdapterWrapper)
            // CoroutineDispatcher defaults to Dispatchers.IO
            .setCoroutineDispatcher(coroutineDispatcher)
            .build()
    }

    override fun onCreate(savedInstanceState: Bundle?) {
```

```
        super.onCreate(savedInstanceState)
        binding = MainActivityBinding.inflate(layoutInflater)
        setContentView(binding.root)
        lifecycleScope.launch {
            lifecycle.repeatOnLifecycle(Lifecycle.State.STARTED) {
                viewModel.viewState.collect(::updateViewState)
            }
        }
        smackSdk.onCreate(this)
    }
}
```

The `DefaultSmackClient` can be changed together with the `NoOpSmackLogger` for testing purposes or to change the logging behavior (see `AndroidSmackLogger`). You may also want to replace `AndroidNfcAdapterWrapper` or the `CoroutineDispatcher` for testing.

Forward NCF intents to the SmackSdk

Incoming NFC `Intent`s must be forwarded to the `SmackSdk` in `onNewIntent()` of your activity. The internal `SmackLifecycleObserver` will forward those events to the smack API classes you use as described in the feature related README documents.

```
class MainActivity : AppCompatActivity() {

    ...

    public override fun onNewIntent(intent: Intent?) {
        super.onNewIntent(intent)
        smackSdk.onNewIntent(intent)
    }
}
```

Minimal example

```
class MainActivity : AppCompatActivity() {

    private val viewModel: MainViewModel by viewModels()
    private lateinit var binding: MainActivityBinding
    private val smackSdk by lazy { SmackSdk.Builder(this).build() }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = MainActivityBinding.inflate(layoutInflater)
        setContentView(binding.root)
        lifecycleScope.launch {
```

```
            lifecycle.repeatOnLifecycle(Lifecycle.State.STARTED) {
                viewModel.viewState.collect(::updateViewState)
            }
        }
        smackSdk.onCreate(this)
    }

    public override fun onNewIntent(intent: Intent?) {
        super.onNewIntent(intent)
        smackSdk.onNewIntent(intent)
    }
}
```

Temporarily enabling/disabling NFC intent processing

In some cases, you want to enable or disable the processing of incoming NFC intents on premise (e.g. a dialog is presented). A `SmackSdk` object provides an `isEnabled` flag you can switch at any time to disable and re-enable the NFC intent processing. If disabled, forwarding an incoming intent to `smackSdk.onNewIntent(intent)` will have no effect.

```
class MainActivity : AppCompatActivity() {

    ...

    private fun showAlert(@StringRes titleRes: Int, @StringRes messageRes: Int) {
        smackSdk.isEnabled = false
        runOnUiThread {
            AlertDialog.Builder(this)
                .setTitle(titleRes)
                .setMessage(messageRes)
                .setNeutralButton(R.string.ok) { dialogInterface, _ ->
                    dialogInterface.dismiss()
                }
                .setOnDismissListener {
                    smackSdk.isEnabled = true
                }
                .show()
        }
    }
}
```

Logging

The `SmackLogger` interface is used for logging the communication between SDK and SmAcK Hardware.

There are 4 log levels:

- ERROR: For exceptions and errors
- INFO: For high level calls like `writeDataPoint()` / `readDataPoint()`
- DEBUG: For low level information about sent and received byte arrays
- VERBOSE: For additional information around the above levels

When initializing a `SmackClient` you can pass your own implementation. By default the NoOpSmackLogger is used, which just omits all logs to avoid that sensitive data is emitted.

If you want to log to logcat, you can use the AndroidSmackLogger implementation. *But warning:* Please keep in mind, that this shouldn't be used in production. Otherwise sensitive information like the lock key can show up in your logs.

Error handling

Various exception can be thrown during the communication with the SmAcK hardware. Exceptions derived from `SmackException` are thrown when a communication error occurs. For more details on error handling check the API methods and it's exception types. All exceptions not derived from `SmackException` are exceptions from the system or transport layer and could be potentially resolved by the user by retrying the operation or restarting the device.

SmackException (abstract)

All exceptions regarding the SmAck protocol are derived from `SmackException`.

SmackException-Type	Description
DataValidationException	This exception is thrown when data is read from or written to the mailbox and a validation fails.
ProtocolException	This exception is thrown on errors in the protocol layer, e.g. unexpected behavior or response from the NFC hardware.
ResponseException	This exception contains the command response (`SmackCommandResponse`) from the response to a sent request. The associated `SmackCommand` should match the command from the request. If not, this exception is raised, containing the mentioned command response.
SmackMailboxException	This exception is thrown on errors regarding the mailbox address or descendant word addresses.

SmackException-Type	Description
HeaderValidationException	This exception contains all header flags (`HeaderFlag`) read from the header word from the mailbox (word at index 1). After the hardware has evaluated the mailbox, the header flags should contain `HeaderFlag.RESPONSE_VALID`. If not, this exception is raised, containing all header flags from the header word, spread into a list for better convenience.
WrongKeyException	This exception is thrown if the hardware can't decrypt the mailbox with the known lock key or supervisor key.
LockIdValidationException	This exception is thrown during the lock setup process in `LockApi.setLockKey(lock, userName, dateTimeInSeconds, supervisorKey, password)`, if the lock ID validation fails.
MailboxMessageValidResponseException	This exception is thrown, if an error occurs on sending a "mailbox message valid" command. It contains the return code (`MailboxMessageValidReturnCode`) from the response to the `MailboxMessageValidRequest`.

Handle errors in the UI

The API classes provide their data using kotlin flows. When you get the flow from an API class, for use in your Activity, Fragment or ViewModel, append a `retry` block to the flow to catch all internal exceptions. The retry must return `true` to keep the flow running indefinitely for subsequent NFC scans. Only for the CancellationException, which cancels the flow, retry must return false. Missing that would keep the flow running forever, even if the related Flow on top is destroyed. If you use `catch` to handle the exception or to emit a value indicating an error, the exception must be rethrown. If you plan to add your own API classes providing flows on top of the ones from this SDK, keep in mind that all exceptions must be rethrown to keep the flows running.

```
someApi
    .getSomeFlow()
    .catch { exception ->
```

```
        // handle the exception, e.g.
        emit(<some error state>)
        // rethrow the exception
        throw exception
    }
    .retry { exception ->
        exception !is CancellationException
    }
```

Note: You may want to **pause NFC processing** after an exception was raised or as long as an error is presented in the UI. To achieve that, disable the SmackLifeCycleObserver by setting `isEnabled = false`. Enable the SmackLifeCycleObserver again, when you're ready to proceed.

SDK Deployment

The compiled SDK .aar file will be published to the x-root Confluence automatically on each push to master. Additionally it will be committed to the git repositories of Showcase and Sample App.

Exporting the AAR manually can be done by running the gradle task `./gradlew sdk:assembleRelease`

Lock Communication

Besides the user oriented functions described here, there is an additional motor control function for developers described under Motor Control

Compatibility with Infineon SmAcK Locks

LockApi is the main entry point for communication with SmAcK NFC locks.

SmackLockApi is an example implementation of lock functions based on a specification defined by Infineon, i.e. work flow, datapoint definition, key management. It works only with a lock design which has the corresponding showcase firmware installed in NAC1080!

If you implemented your own Smart Lock Firmware, you can provide your own implementation of LockApi.

Lock

Obtaining a lock object

First you can obtain a lock by collecting the LockApi's getLock()-Flow. A Lock-Object will be generated for every physical connection to a SmAcK Smart Lock and then sent through the StateFlow. When the connection gets lost, the Flow will emit a null-Lock.

Each lock carries information about the unique `lockId` and whether a lock key was initialized before or not (`isNew`).

Validate password / Create a lock key

Performing actions on a Lock is only permitted with a valid lock key. To obtain a lock key from a password, you need to call `LockApi.validatePassword()`. This will generate a lock key from the password, initialize a session with this lock key to validate it with the lock and then return it for you to store.

Providing a wrong `password` to this function will result in a WrongKeyException

After obtaining a lock key you can save it for later use in the next session. See Initializing a session for details how to control a lock with a known lock key.

Initializing a session

Before you can perform any actions on the `Lock`, you need to authenticate by calling `LockApi.initializeSession()`. This initializes the Session by choosing the SessionType SessionType.USER and setting all relevant information for Lock History.

Providing a wrong `lockKey` to this function will result in a WrongKeyException

Setting up a brand new lock or changing a lock password

Before the first usage, brand new locks must be setup with a supervisor key and a password to generate a lock key. Ask the user for a password and the supervisor key and call `LockApi.setLockKey(lock, userName, dateTimeInSeconds, supervisorKey, password)` with this data to generate and set a new lock key for the lock. You can find the supervisor key in the locks original packaging. After setting up the lock, the newly generated `LockKey` is used for further communication. You can use the exact same procedure for changing a lock password.

Performing actions

After obtaining a lock object and initializing the session you can finally perform actions on the lock.

With LockApi the following lock functions are available:

LockApi	Parameters	Response	Description
getChargeLevel	lock	ChargeLevel	Read the current level of charging
lock	lock	boolean	Sends a locking request

LockApi	Parameters	Response	Description
unlock	lock	boolean	Sends an unlocking request
getLockHistory	lock	List	Requests the access history of the lock

Charging Level

Please note that the charging level can constantly change with every bit of communication that happens between Smartphone and Lock. You have to use LockApi to query the current level of charging.

Technical Background

Mailbox Each Lock carries a Mailbox object for the lower level of communication, in particular: reading and writing Datapoints. (See Mailbox Communication)

See `com.infineon.smack.sdk.application.lock.datapoint.LockDataPoint` for a list of available Lock-Datapoints.

User Modes See SessionType as a reference.

SessionType.USER -Mode is used for normal operation. This is set automatically when calling `LockApi.initializeSession()`.

SessionType.SUPERVISOR -Mode is used for setting a new lockKey. This is set automatically when calling `LockApi.setLockKey()`.

New lock setup and change lock password procedure In the background the following procedure is executed when calling `LockApi.setLockKey()`:

- set the session type to `SessionType.SUPERVISOR`
- set the access date for logging
- set the access user name for logging
- generate a `LockKey` from the lock ID and the given password
- write the newly generated `LockKey` to the lock
- read and validate the lock ID encrypted with tne newly generated `LockKey`
- store the newly generated `LockKey` in the lock for further usage

For this procedure the communication is encrypted with the supervisor key.

Examples

Get a connection and use the lock To communicate with the lock, you first need a `LockApi` with a client. `SmackLockApi` provides a default implementation.

The `DefaultSmackClient` can be used to communicate with supported NFC hardware. Then the `getLock()` of the `LockApi` must be observed. This `lock` is designed with the asynchronous `Flow` component from kotlin coroutines. A good approach is to implement a view model and convert the `Flow` into `StateFlow` that can be collected from the `Fragment`. As soon as you hold your device onto the NFC hardware, the connection will be established, the smack protocol started, the mailbox address requested and the lock id queried, so everything is set up to communicate with the lock.

```
class MainViewModel : ViewModel(smackSdk: SmackSdk) {

    private val lockApi = smackSdk.lockApi

    val viewState: StateFlow<MyViewState> = lockApi.getLock()
        .map { lock ->
            val knownLockKey = getLockKeyFromDatabase(lock)
            if (lock != null && knownLockKey != null) {
                lockApi.initializeSession(
                            lock,
                            "MyUserName",
                            System.currentTimeMillis() / 1000,
                            knownLockKey
                        )
                do {
                    val chargeLevel: ChargeLevel = getChargeLevel(lock)
                } while (!chargeLevel.isFullyCharged)
                lockApi.unlock(lock)
            } else if (lock != null && knownLockKey == null) {
                if (lock.isNew) { showSetupLockScreen() }
                // Show a register screen to setup a brand new lock (See example below)
                else { showEnterLockPasswordScreen() }
                // Show a login screen to setup a lock that was already registered before (S
            }
            ...
        }
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(0, 0),
            initialValue = MyViewState()
        )
}
```

Setup a brand new lock or change the lock password

```
class RegistrationViewModel : ViewModel(smackSdk: SmackSdk) {
```

```
    private val lockApi = smackSdk.lockApi
    private val userName = "User Name"
    private val supervisorKey = "fedcba9876543210"
    private val newPassword = "0123456789abcdef"

    val viewState: StateFlow<MyViewState> = lockApi.getLock()
        .map { lock ->
            if (lock != null && lock.isNew) {
                val lockKey = lockApi.setLockKey(
                    lock,
                    userName,
                    LocalDateTime.now().toEpochSecond(ZoneOffset.UTC),
                    supervisorKey,
                    newPassword
                )
                // Store the lock key for this lock, for further usage.
            }
            ...
        }
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(0, 0),
            initialValue = MyViewState()
        )
}
```

Setup a lock that was already registered before To generate a lock key that can be used to control the lock, the user has to enter a password. This password needs to be transformed into a lock key by calling `LockApi.validatePassword()`.

```
val passwordTheUserEntered = "1234"
lockApi.getLock().map { lock ->
    // This block gets called with a non-null lock object for every physical connection
    if (lock != null && !lock.isNew) {
        val lockKey = lockApi.validatePassword(
                        lock,
                        "MyUserName",
                        System.currentTimeMillis() / 1000,
                        passwordTheUserEntered
                    )
        saveLockKeyToDatabase(lock, lockKey)
    }
}
```

Mailbox Communication

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.

Following functions are available:

MailboxApi	Parameters	Response	Description
readWord	mailbox, index	word	Read one word from the mailbox at the given index
writeWord	mailbox, word, index	smack response	Write one word to the mailbox at the given index
readDataPoint	mailbox, data point, length of dynamic data if applicable (optional), encryption key (optional)	words	Read data with a given length from a data point
writeDataPoint	mailbox, data point, data to write, encryption key (optional)	list of smack responses	Write data to a data point
getFirmwareName	mailbox, encryption key	the firmware name as a String	Read the firmware name
getFirmwareVersion	mailbox, encryption key	the firmware version as a String	Read the firmware version
getChargeLevel	mailbox, encryption key	the charge level of the nfc hardware	Read the charge level
callAppFunction	mailbox, app function index, data to write at indices 1 to n	the first 64 bytes of the mailbox content	Call an app function

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 (`MailboxDataPoint`) 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: 'call app function', 'mailbox header' and 'data point', then comes the 'data' requests which are split up into chunks per word and lastly the request 'mailbox message valid' (`MailboxMessageValidRequest`) which advises the firmware to calculate its state according to the parameters set before. The first four + n requests are summarized in `DataPointRequestList`. Validating a data point takes two steps: first reading the header word and checking bitwise whether its flags (last two bytes) indicate a valid response, second reading the data point word and comparing it with the data that was being written before.

Data Point Types Each data point has its own data type. And each data type has a defined length, except two: array and string data types have a dynamic length. So when reading a data point with one of these data types, the length of the data to read must be provided.

Examples

Get a connection to the mailbox To communicate with the mailbox, you first need a `MailboxApi` with a client. The `DefaultSmackClient` can be used to communicate with supported NFC hardware. Then the `mailbox` of the `MailboxApi` must be observed. This `mailbox` is designed with the asynchronous `Flow` component from kotlin coroutines. A good approach is to implement a view model and convert the `Flow` into `StateFlow` that can be collected from the `Fragment`. As soon as you hold your device onto the NFC hardware, the connection will be established, the smack protocol started and the mailbox address requests, so everything is set up to communicate with the mailbox.

```
class MainViewModel : ViewModel(smackSdk: SmackSdk) {

    private val mailboxApi = smackSdk.mailboxApi

    val viewState: StateFlow<MyViewState> = mailboxApi.mailbox
        .map { mailbox ->
            // do something with the mailbox
            ...
        }
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(0, 0),
```

```
            initialValue = MyViewState()
        )
}
```

Reading a data point Assuming you already observe the mailbox as described above, the mailbox function `readDataPoint(SmackMailbox, MailboxDataPoint, Byte)` can be called to read a data point.

```
mailboxApi.readDataPoint(mailbox, MailboxDataPoint.TEMPERATURE)
```

Note: The last parameter from `readDataPoint(SmackMailbox, MailboxDataPoint, Byte)` `dynamicDataToReadLength` was omitted in this example. Providing the length of the data to read is only necessary for data points with a dynamic length, like DataPointType.STRING or DataPointType.ARRAY. If you miss that, a `SmackException` will be thrown to indicate the error.

Writing a data point Assuming you already observe the mailbox as described above, the mailbox function `writeDataPoint(SmackMailbox, MailboxDataPoint, Byte)` can be called to write to a data point.

```
mailboxApi.writeDataPoint(mailbox, MailboxDataPoint.SCRATCH16, data)
```

Note: The length of the provided data must match the data type of the given data point. In this example the data type of SCRATCH16 is UINT16, which is two bytes long. A `SmackException` is thrown, if the data type and the length of the provided data don't match.

Reading the firmware info

```
val firmwareName = mailboxApi.getFirmwareName(mailbox, lockKey)
val firmwareVersion = mailboxApi.getFirmwareVersion(mailbox, lockKey)
```

Calling an app function

```
val responseBytes =
        mailboxApi
            .callAppFunction(mailbox, 15, RandomByteArrayFactory.createWord())
            .joined()
```

Measurement

The `MeasurementApi` provides access to request certain measurement data from a device. Use the SmackSdk to get both a `MailboxApi` and a `MeasurementApi` and observe the `mailbox` `Flow` of the `MailboxApi`. You'll need the `mailbox` to request measurement data.

Compatibility with Infineon NAC1080

MeasurementApi is the main entry point for communication with SmAcK NFC measurement hardware.

SmackMeasurementApi is an example implementation of measurement functions based on a specification defined by Infineon, i.e. work flow, datapoint definition, key management. It works only with a lock design which has the corresponding showcase firmware installed in NAC1080!

If you implemented your own Measurement Firmware, you can provide your own implementation of MeasurementApi.

Functions

Following functions can be executed via the MeasurementApi.

Note: To make sure that the sensors have finished their measurement, requests may be retried up to 3 times. This is handled by the MeasurementApi and does not affect the way of using it.

MeasurementApi	Parameters	Response	Description
getTemperature	mailbox	Float	Retrieve the temperature of the currently connected hardware in °C
getHumidity	mailbox	Float	Retrieve the relative humidity of the currently connected hardware in %
getPressure	mailbox	Float	Retrieve the pressure of the currently connected hardware in bar
getReserved	mailbox	Int	Retrieve a value reserved for future use of the currently connected hardware

Examples

Initialize the MeasurementApi and read temperature

To communicate with the measurement hardware, you first need a `MailboxApi` with a client. The `DefaultSmackClient` can be used to communicate with supported NFC hardware. Then the `mailbox` of the `MailboxApi` must be observed. This `mailbox` is designed with the asynchronous `Flow` component from kotlin coroutines. A good approach is to implement a view model and convert the `Flow` into `StateFlow` that can be collected from the `Fragment`. As soon as you hold your device onto the NFC hardware, the connection will be established, the smack protocol started and the mailbox address requests, so everything is set up to communicate with the mailbox.

```
class MainViewModel : ViewModel(smackSdk: SmackSdk) {

    private val mailboxApi = smackSdk.mailboxApi
    private val measurementApi = smackSdk.measurementApi

    val viewState: StateFlow<MyViewState> = mailboxApi.mailbox
        .map { mailbox ->
            val temperature = measurementApi.getTemperature(mailbox)
        }
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(0, 0),
            initialValue = MyViewState()
        )
}
```

Motor Control

The motor control API is part of the `LockApi` but shouldn't be used in end user apps. By writing motor control parameters, the behavior of the motor can be modified.

Motor control parameters

There are three different `MotorControlParameters` to choose from. Use the static functions to construct the one you need:

- `MotorControlParameters.forSingleMovement()`: includes a start voltage in millivolt
- `MotorControlParameters.forVoltageControlled()`: includes a start and a stop voltage in millivolt
- `MotorControlParameters.forTimerControlled()`: includes an on and off time in milliseconds

Note that all construction functions need a `ClampingVoltage` and a total motor runtime, but other parameters depend on the configuration method (`ConfigMethod`). See the class documentation for more restrictions on the parameters.

Write motor control parameters

Create the desired motor control parameters using a static function of `MotorControlParameters` as described above. Use `LockApi.writeMotorControlParameters(lock, parameters, lockKey)`. Motor control parameters are encrypted, so a lock key is necessary. See Lock Communication for more info on how to get a lock key.

Example

```
class MotorControlViewModel : ViewModel(smackSdk: SmackSdk) {

    private val lockApi = smackSdk.lockApi
    private val clampingVoltage = ClampingVoltage.MEDIUM
    private val totalMotorRuntime = 2000u
    private val startVoltage = 1000u
    private val stopVoltage = 2000u

    val viewState: StateFlow<MyViewState> = lockApi.getLock()
        .map { lock ->
            val knownLockKey = getLockKeyFromDatabase(lock)
            if (lock != null && knownLockKey != null) {
                lockApi.initializeSession(
                    lock,
                    "MyUserName",
                    System.currentTimeMillis() / 1000,
                    knownLockKey
                )
                lockApi.writeMotorControlParameters(
                    lock,
                    clampingVoltage,
                    totalMotorRuntime,
                    MotorControlParameters.forVoltageControlled(
                        clampingVoltage,
                        totalMotorRuntime,
                        startVoltage,
                        stopVoltage
                    ),
                    knownLockKey
                )
            }
            ...
```

```
        }
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(0, 0),
            initialValue = MyViewState()
        )
}
```