ALIEN TECHNOLOGY®

ALR-S350 (Sled) Developer Guide

April, 2018

ALIEN®

ALR-S350

Legal Notices

Alien Technology, LLC and/or its affiliated companies have intellectual property rights relating to technology embodied in the products described in this document, including without limitation certain patents or patent pending applications in the U.S. or other countries.

This document and the products to which it pertains are distributed under licenses restricting their use, copying, distribution and decompilation. No part of this product documentation may be reproduced in any form or by any means without the prior written consent of Alien Technology, LLC and its licensors, if any. Third party software is copyrighted and licensed from Licensors. Alien, Alien Technology, the Alien logo, Nanoblock, Fluidic Self Assembly, FSA, Gen2Ready, Squiggle, Nanoscanner and other graphics, logos, and service names used in this document are trademarks of Alien Technology, LLC and/or its affiliated companies in the U.S. and other countries. All other trademarks are the property of their respective owners. U.S. Government approval required when exporting the product described in this documentation.

Federal Acquisitions: Commercial Software -- Government Users Subject to Standard License Terms and Conditions. U.S. Government: If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

DOCUMENTATION IS PROVIDED “AS IS” AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARANTEES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGMENT ARE HEREBY DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Alien Technology®
ALR-S350 Developer Guide

Table of Contents

1 Introduction

The Alien ALR-S350 Software Developer Kit (SDK) provides libraries and sample code to programmatically control Alien ALR-S350 handheld readers from either Android OS or iOS based mobile devices. Alien provides class libraries and sample applications to help you get up-and-running developing your custom applications.

The purpose of this guide is to provide information about key basic steps required to create an app communicating with the ALR-S350 device.

The API has been developed for Alien Technology by its OEM partner Nordic ID. For more detailed information, including source code of fully featured demo apps, refer to the Nordic ID github based repository at https://github.com/NordicID

Android:
https://github.com/NordicID/nur_sample_android/
iOS:
https://github.com/NordicID/nur_sample_ios

1.1 Audience

We assume that the readers of this guide:

- are proficient Android or iOS developers,
- have minimal previous knowledge of RFID, and other relevant technologies.

1.2 Type Conventions

- Regular text appears in a plain, sans-serif font.
- External files and documents appear in *italic text*.
- Class names appear in a `fixed-width serif font`.
- Things you type in, and sample code appear:

```
indented, in a fixed-width serif font.
```

- Longer blocks of sample code appear like below:

```
// start inventory stream
   mApi.startInventoryStream();
```

1.3 Overview

This document explains how to programmatically control the ALR-S350 handheld readers using the provided APIs.

1.4 Documentation and Sample Code

This Guide provides high level overview of the API elements. Additional usage tips can be gleaned by examining and modifying the source code samples included in the SDK.

1.5 System Requirements

You will need Android Studio (recommended version is 3.0+) to develop applications running on Android OS, or Xcode (recommended version is 9+) for iOS based apps.

2 Developing Android applications

2.1 Install Android Studio

In order to develop RFID applications for the Alien ALR-S350 handheld, install Android Studio (version 3.0+ is recommended) on your computer.

Download Android Studio from http://developer.android.com/sdk/index.html

Run the installer using all the default settings. The installer will automatically download and install the required components, including Android Support Repository and Android SDK Tools.

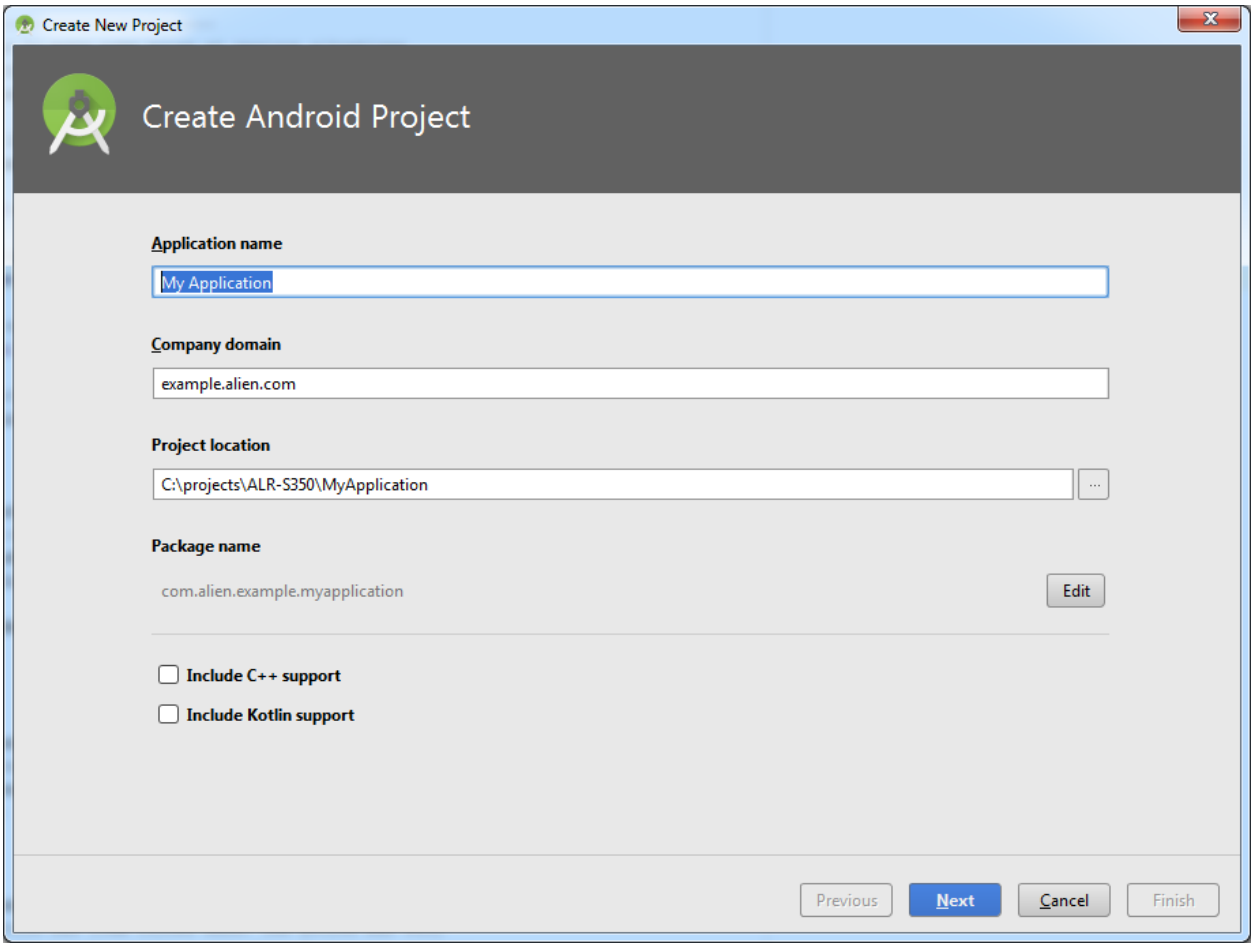
Make sure to install Google USB drivers as well as enable Developer Mode and check “USB Debugging” in “Developer options” on your mobile device.

Now you are ready to develop applications for the handheld.

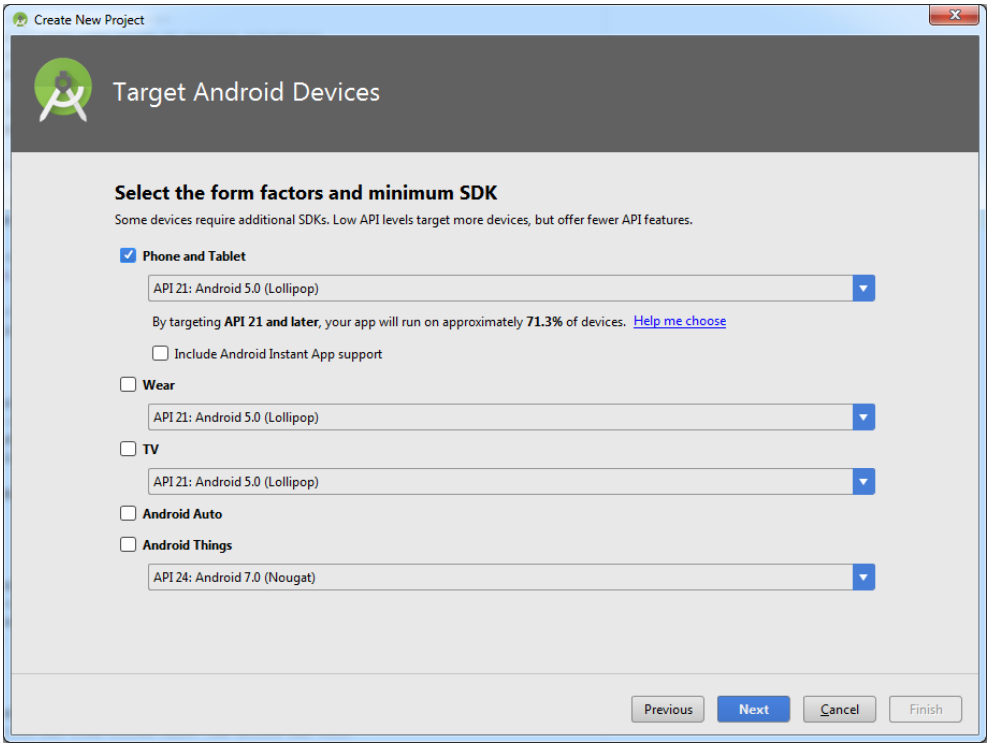
2.2 Adding the API to your application

2.2.1 Create Android Project

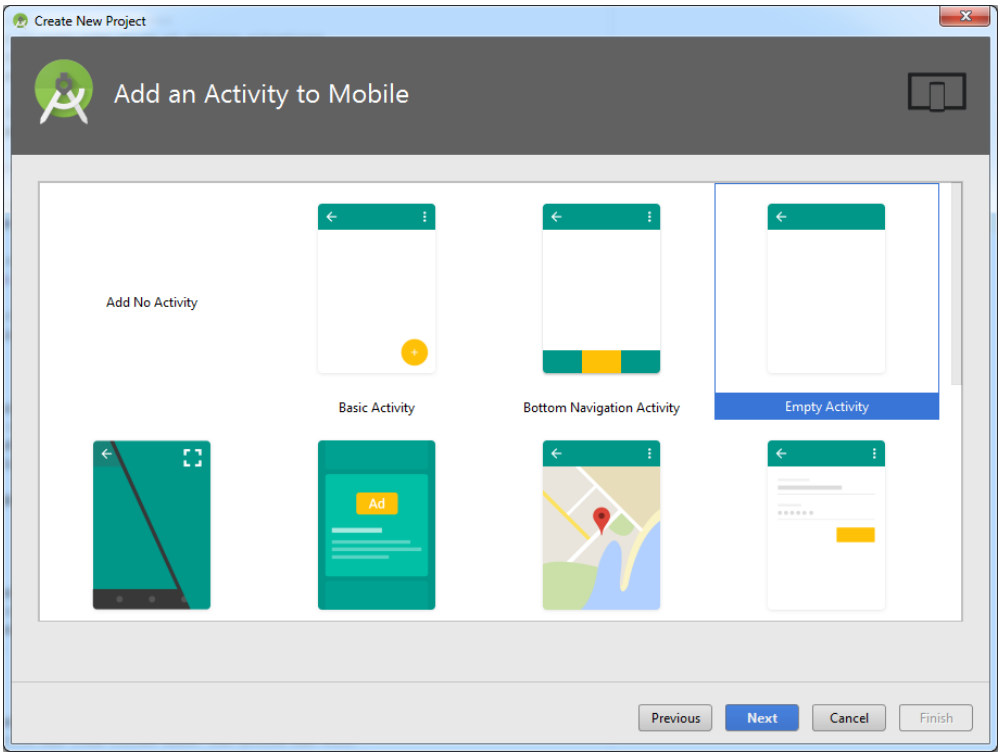
Open Android Studio and create a new project.



In the “Target Android Devices” dialog, check the “Phone and Tablet” checkbox and select “API21: Android 5.0 (Lollipop)” as the Minimum SDK. Click “Next”:



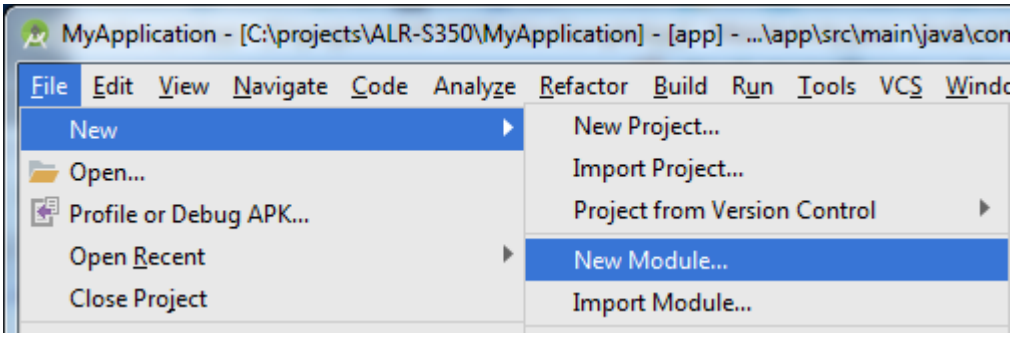
In the “Add an activity to Mobile” dialog, select “Empty Activity” and click “Next”, then either use the defaults or enter new names for Activity Name, Layout Name, Title and Menu Resource Name. Click “Finish”:



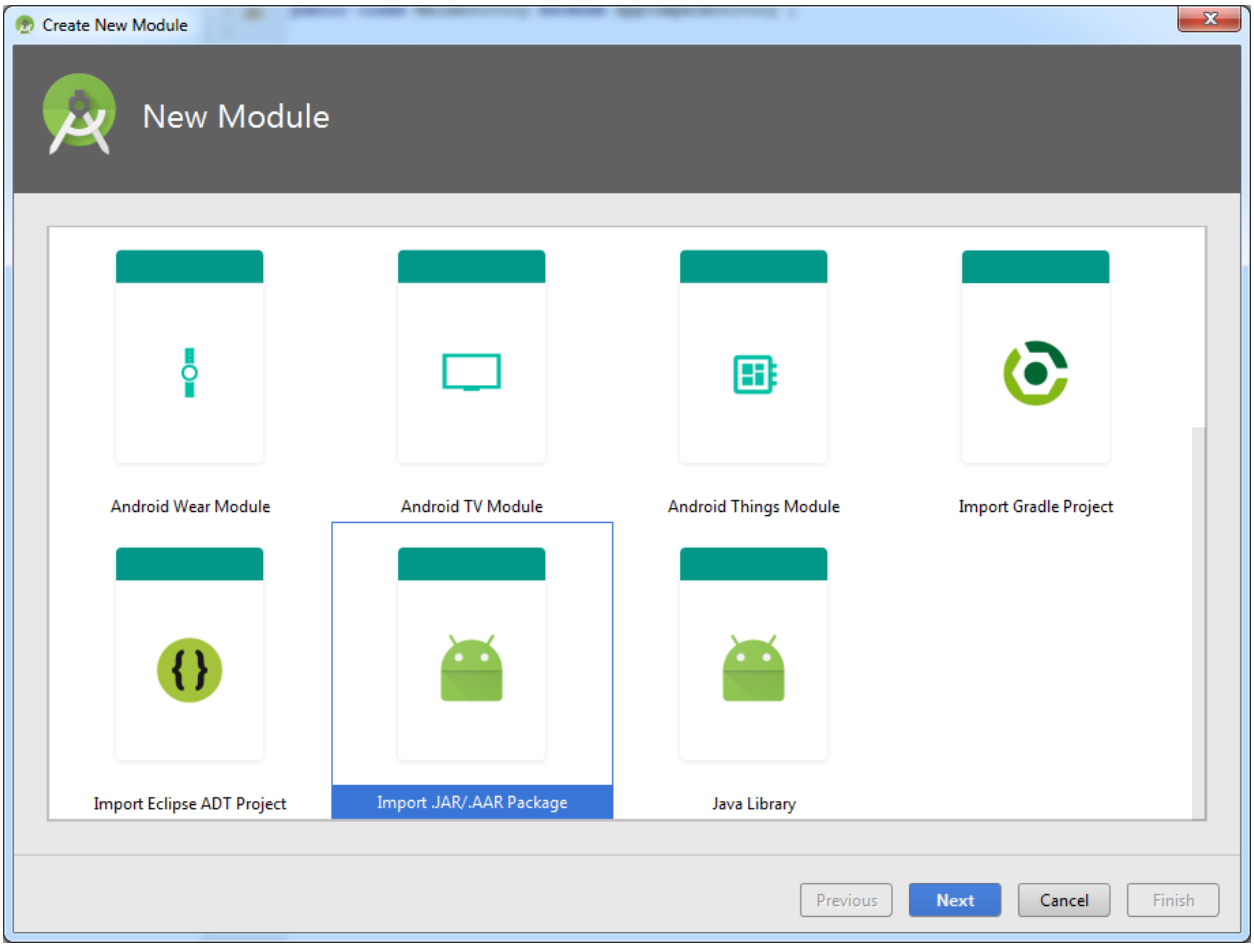
2.2.2 Add API modules to your project

In order to communicate with the device, add the `NurApi.jar` and `NurApiAndroid.aar` libraries to your project as new modules.

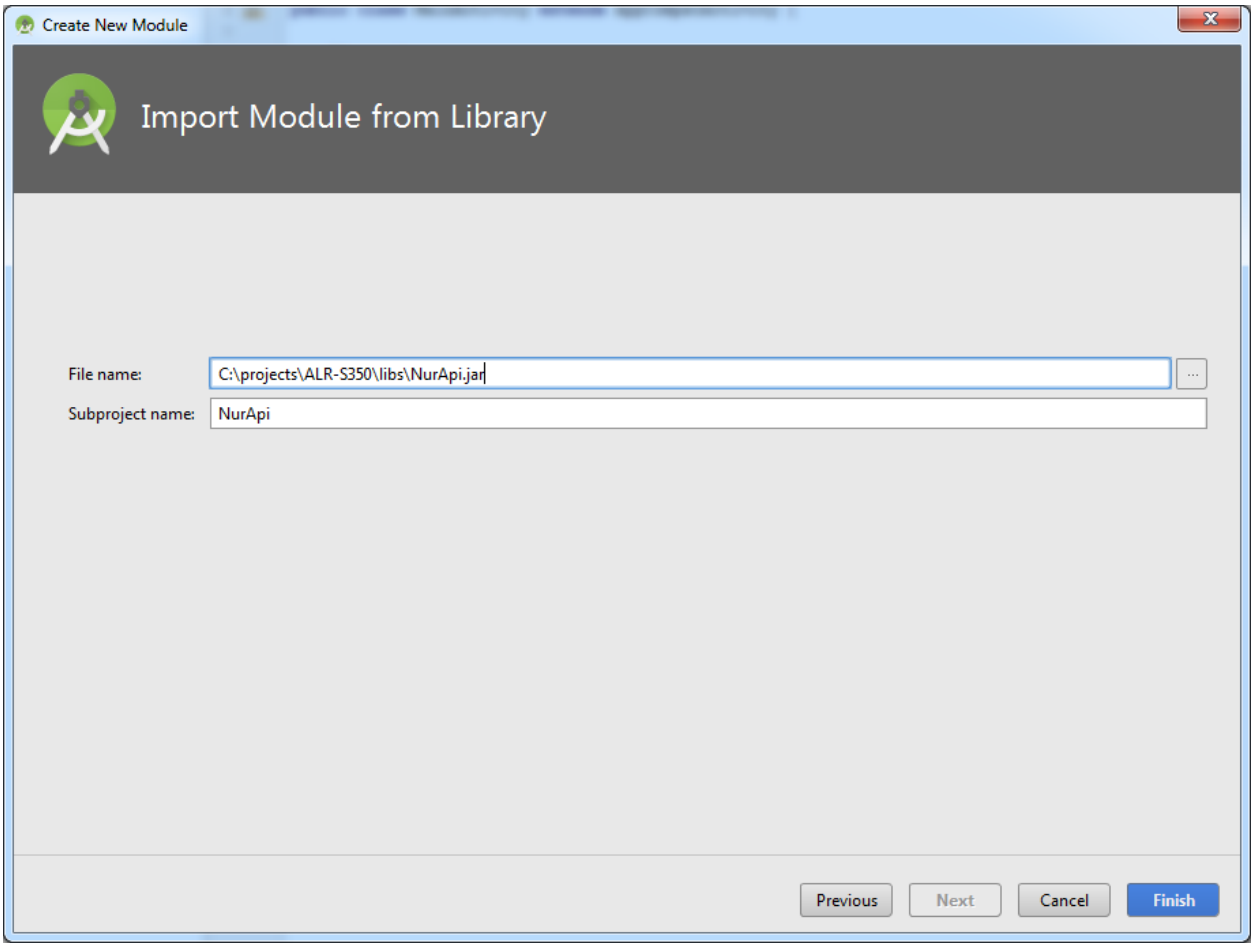
Select menu: File > New > New Module:



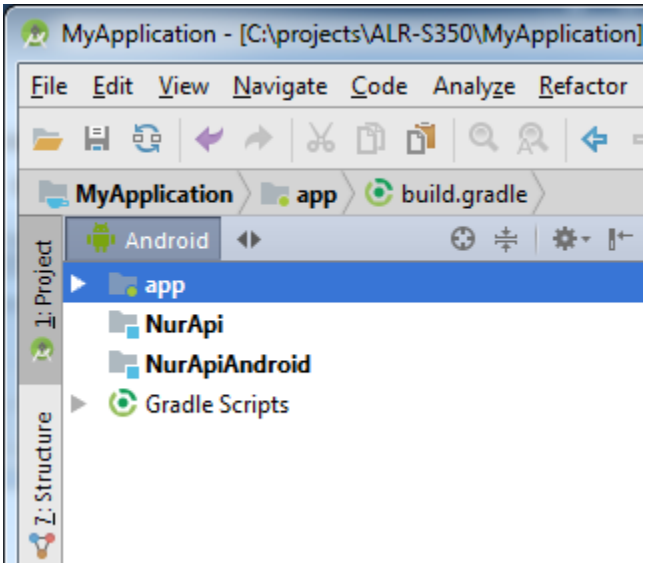
In the “New Module” dialog, select “Import .JAR/.AAR Package” and click “Next”:



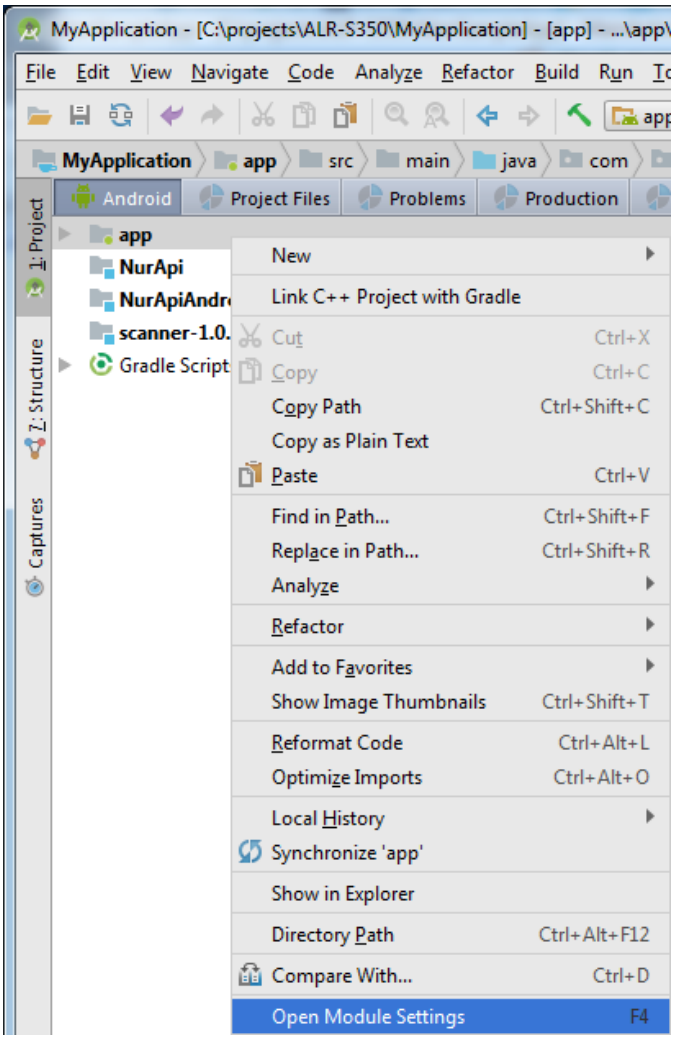
Browse to select the `NurApi.jar` library file and click “Finish”:

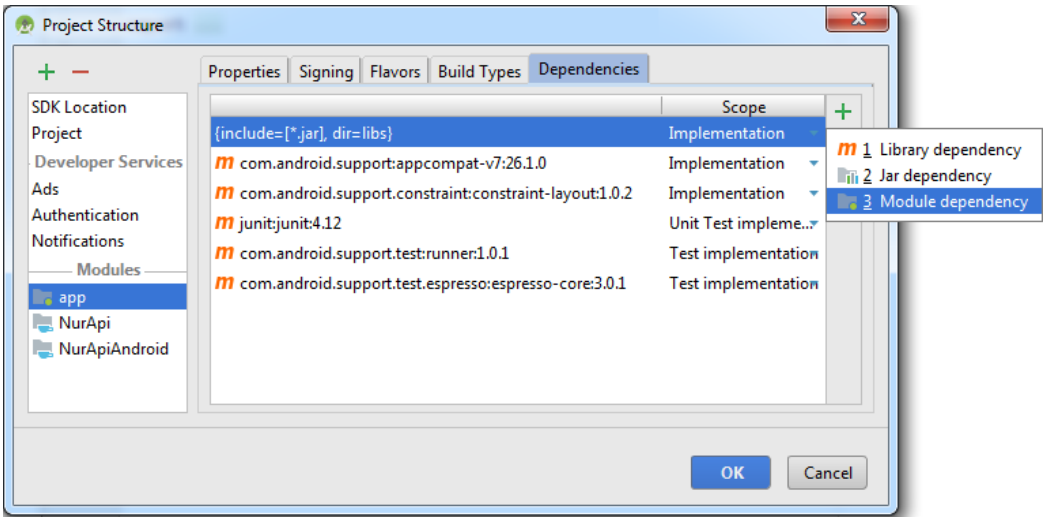


Add `NurApiAndroid.aar` library in a similar fashion and now you should see both “**`NurApi`**” and “**`NurApiAndroid`**” modules added to your project:

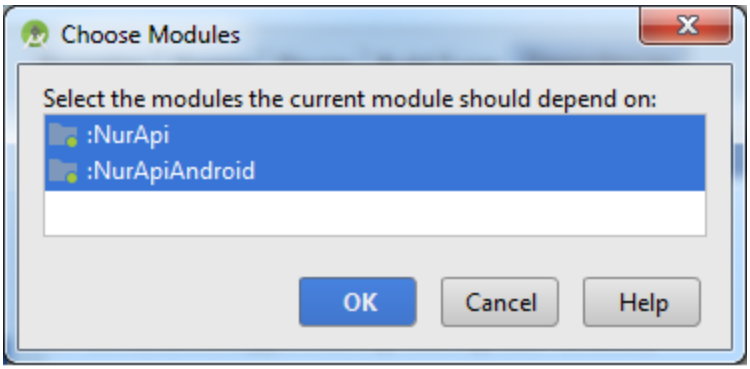


Next, configure the "**app**" module to use "**NurApi**" and "**NurApiAndroid**" as dependencies:
- Right click on the "app" module, and select "**Open Module Settings (F4)**".
- Select the "app" module, select the "**Dependencies**" tab, click "**+**" button and select "**Module dependency**".



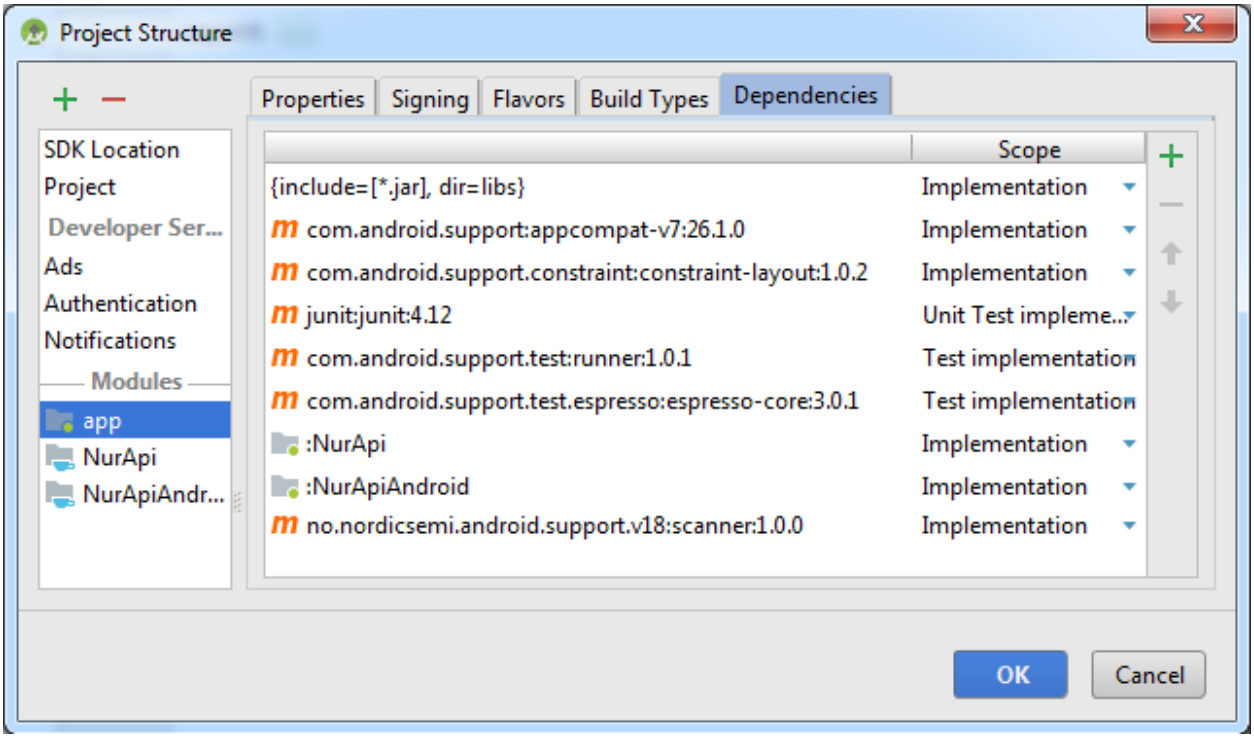


Select both "**NurApi**" and "**NurApiAndroid**" and click "OK"



Click the "**+**" button again, select "**Library dependency**" and enter "`no.nordicsemi.android.support.v18:scanner:1.0.0`"

Now you should see "**`NurApi`**" and "**`NurApiAndroid`**" as well as "`no.nordicsemi.android.support.v18:scanner:1.0.0`" in the Dependencies list.



The presense of the required dependencies can be verified by opening the `build.gradle (Module app)`:

```
dependencies {
...
    implementation project(':NurApiAndroid')
    implementation project(':NurApi')
    implementation 'no.nordicsemi.android.support.v18:scanner:1.0.0'
}
```

2.2.3 Modify AndroidManifest.xml

Modify the `AndroidManifest.xml` file to add:

- User `permissions` for Bluetooth
- `UartService`
- `NurDeviceListActivity` declaration to be used for device discovery

```
<manifest package="com.alien.example.myapplication"
        xmlns:android="http://schemas.android.com/apk/res/android">
...
   <uses-permission android:name="android.permission.BLUETOOTH" />
   <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
   <uses-permission android:name="android.permission.BLUETOOTH_PRIVILEGED" />

   <application
   ...
      <service
          android:name="com.nordicid.nurapi.UartService"
          android:enabled="true"
          android:exported="true" />

      <activity
          android:name="com.nordicid.nurapi.NurDeviceListActivity"
          android:label="@string/app_name"
          android:theme="@android:style/Theme.Dialog"/>
      ...
   </application>
...
</manifest>
```

2.3 Using the API in your application

The API consists of two libraries:

- `NurApi.jar` Java library for RFID and BLE (Bluetooth Low Energy) transport operations. The transport layer handles the low level communication with the sled module
- `NurApiAndroid.aar` library for Android includes extensions and device specific operations such as Barcode reading, BLE connectivity, battery status, I/O events etc. The `NurApiAndroid` library also includes an auto-connect interface (`NurApiAutoConnectTransport`) to simplify re-connection to a known device (identified by it's Bluetooth MAC address)

2.3.1 Add API imports

Add the following imports to your app source code:

```
import com.nordicid.nuraccessory.*;
import com.nordicid.nurapi.*;
```

2.3.2 Declare API instance

Declare the API instance and, optionally, the accessory extension. The latter is required to interface with the barcode scanner or obtain battery information and takes the API as its parameter:

```
private NurApi mApi = new NurApi();
private NurAccessoryExtension mAccessoryApi = new NurAccessoryExtension(mApi);
```

2.3.3 Add API event handlers

In order to receive events from `NurApi`, `NurApiListener` must be defined. Event handlers must be defined for all possible events the `NurApi` may fire.

Each activity must have its own `NurApiListener` and event handlers.

```
// set mNurApiListener to listen for API events
mApi.setListener(mNurApiListener);
```

```
private NurApiListener mNurApiListener = new NurApiListener()
{
    @Override
    public void triggeredReadEvent(NurEventTriggeredRead event) { }
    @Override
    public void traceTagEvent(NurEventTraceTag event) { }
    @Override
    public void programmingProgressEvent(NurEventProgrammingProgress event) { }
    @Override
    public void nxpEasAlarmEvent(NurEventNxpAlarm event) { }
    @Override
    public void logEvent(int level, String txt) { }
    @Override
    public void inventoryStreamEvent(NurEventInventory event) { }
    @Override
    public void inventoryExtendedStreamEvent(NurEventInventory event) {}
    @Override
    public void frequencyHopEvent(NurEventFrequencyHop event) { }
    @Override
    public void epcEnumEvent(NurEventEpcEnum event) { }
    @Override
    public void disconnectedEvent() { }
    @Override
    public void deviceSearchEvent(NurEventDeviceInfo event) { }
    @Override
    public void debugMessageEvent(String event) { }
    @Override
    public void connectedEvent() { }
    @Override
    public void clientDisconnectedEvent(NurEventClientInfo event) { }
    @Override
    public void clientConnectedEvent(NurEventClientInfo event) { }
    @Override
    public void bootEvent(String event) {}
    @Override
    public void IOChangeEvent(NurEventIOChange event) { }
    @Override
    public void autotuneEvent(NurEventAutotune event) { }
    @Override
    public void tagTrackingScanEvent(NurEventTagTrackingData event) { }
    //@Override
    public void tagTrackingChangeEvent(NurEventTagTrackingChange event) { }
};
```

Since all `NurApi` events are called from `NurApi` thread, direct UI updates are not allowed. Use, for example, `runOnUiThread(Runnable)` if you need to access UI elements:

```
private void updateUI() {
   runOnUiThread(new Runnable() {
      @Override
      public void run() {
         btnConnect.setText(txtConnect);
      }
   });
}

...


private NurApiListener mNurApiListener = new NurApiListener()
{
...
   @Override
   public void connectedEvent() { // Device is connected
      txtConnect = "DISCONNECT";
      updateUI();
   }

   public void disconnectedEvent() {
      txtConnect  = "CONNECT";
      updateUI();
   }
...
}
```

2.3.4 Device discovery and selection

Reader devices are advertised via Bluetooth LE. Use `NurDeviceListActivity` to show and select devices from the list of discovered devices.

Ensure `NurDeviceListActivity` and `UartService` are defined in `AndroidManifest.xml`:

```
      <service
          android:name="com.nordicid.nurapi.UartService"
          android:enabled="true"
          android:exported="true" />
      <activity
          android:name="com.nordicid.nurapi.NurDeviceListActivity"
          android:label="@string/app_name"
          android:theme="@android:style/Theme.Dialog"/>
      ...
   </application>
```

Then call the device discovery and selection activity from your app as follows:

```
NurDeviceListActivity.startDeviceRequest(MainActivity.this, mApi);
```

The result of the device list activity is handled as follows:

The `requestCode` code we are interested in is "`NurDeviceListActivity.REQUEST_SELECT_DEVICE`" (`32778`) and if the `resultCode` is `RESULT_OK` then the user has selected the device.
Upon selecting the device create `NurDeviceSpec` and `NurApiAutoConnectTransport` auto-connect transport.

```
@Override
public void onActivityResult(int requestCode, int resultCode, Intent data)
{
   switch (requestCode)
   {
      case NurDeviceListActivity.REQUEST_SELECT_DEVICE: {
         if (data == null || resultCode != NurDeviceListActivity.RESULT_OK)
            return;

         try {
            NurDeviceSpec spec = new
                NurDeviceSpec(data.getStringExtra(NurDeviceListActivity.SPECSTR));

         // dispose existing auto-connect transport
            if (hAcTr != null)
                hAcTr.dispose();

            hAcTr = NurDeviceSpec.createAutoConnectTransport(this, mApi, spec);
            String strAddress = spec.getAddress();
            hAcTr.setAddress(strAddress);

         // in order to connect to the same device automatically on startup, save
         // 'strAddress" and use it to re-connect on the app start
            saveSettings(spec);

         } catch (Exception e) {
            e.printStackTrace();
         }
      }
      break;
   }
   super.onActivityResult(requestCode,resultCode,data);
}
```

After calling `spec.setAddress`, connection will be established and maintained automatically. Use auto-connect transport `onStop, onPause, onResume, onDestroy` to control the connection.

```
@Override
protected void onPause() {
   super.onPause();

   if (hAcTr != null) hAcTr.onPause();
}

@Override
protected void onResume() {
   super.onResume();
   if (hAcTr != null) hAcTr.onResume();
}

@Override
protected void onStop() {
   super.onStop();
   if (hAcTr != null) hAcTr.onStop();
}

@Override
protected void onDestroy() {
   super.onDestroy();
   if (hAcTr != null) hAcTr.onDestroy();
}
```

2.3.5 Performing tag inventories

Use `startInventoryStream()` method to start inventory streaming that results in `InventoryStream` events. Inventory stream is active for about 20 sec and then stopped automatically but it could be immediately restarted after receiving the inventory `stopped` event.

```
private void onStartInventory()
{
   try {
// clear tag data currently stored in module's memory and the API's internal storage
      mApi.clearIdBuffer();

// start inventory stream. Now inventoryStreamEvent handler handles inventory results
      mApi.startInventoryStream();
   }
   catch (Exception ex) { }
}

private void onStopInventory()
{
   try {
      if (mApi.isInventoryStreamRunning())
         mApi.stopInventoryStream();
   }
   catch (Exception ex) { }
}
```

2.3.6 Handling inventory stream events

```
private NurApiListener mNurApiEventListener = new NurApiListener()
{
   ...
       @Override
      public void inventoryStreamEvent(NurEventInventory event) {
         try {
            if (event.stopped) {
            // InventoryStreaming automatically stops after ~20 sec but
            // it can be restarted immediately if needed
            // check if need to restart
               if (inventoryActive) // Trigger button is still down so start it again
                  mApi.startInventoryStream();

            } else {
               if(event.tagsAdded > 0) {
               // at least one new tag found
                  if(MainActivity.IsAccessorySupported())
                     mAccessoryApi.beepAsync(20); // beep on device

               // Tag Storage contains all found tags
                  NurTagStorage tagStorage = mApi.getStorage();

               // move tags from NurApi tag storage to our tagList
                  for(int i=0; i<tagStorage.size(); i++) {
                  // get a tag from storage
                     NurTag tag = tagStorage.get(i);
                     addTag(tag);
                     tagCount++;
//                   String epc = NurApi.byteArrayToHexString(tag.getEpc());
                  }

                  tagStorage.clear(); // clear NurApi tag storage
                  updateUI();
               }
            }
         }
         catch (Exception ex)
         {
            txtStatus = ex.getMessage();
            colorStatus = Color.RED;
            updateUI();
         }
      }
   ...
};
```

2.3.7 Handling I/O events

Triggering the device buttons generates `NurEventIOChange` events. The '`source`' and '`direction`' values are used to determine which button changes state as follows.

Source

- 100 = Trigger button
- 101 = Power button
- 102 = Unpair button

Direction

- 0 = button released
- 1 = button pressed down

```
private void HandleIOEvent(NurEventIOChange event)
{
   try {
      switch (event.source) {
         case 100: // Trigger down
            if(event.direction == 1)
               StartInventory(); // Trigger pulled down. Start Inventory
            else
               StopInventory(); // Trigger released. Stop inventory.
            break;
         case 101: // Power button pressed or released
            break;
         case 102: // Unpair button pressed or released
            break;
         default:
            break;
      }
   }
   catch (Exception ex)
   {
      txtStatus = ex.getMessage();
   }

   updateUI();
}


private NurApiListener mNurApiEventListener = new NurApiListener()
{
   ...
       @Override
   public void IOChangeEvent(NurEventIOChange event) {
      HandleIOEvent(event);
   }
   ...
};
```

2.3.8 Reading barcodes

Similar to setting up the event listeners for RFID operations, register the API event listener and barcode operation result listener:

```
// set event listener for this activity
mApi.setListener(mNurApiEventListener);
mAccessoryApi.registerBarcodeResultListener(mResultListener);

...

private AccessoryBarcodeResultListener mResultListener =
                    new AccessoryBarcodeResultListener()
{
   @Override
   public void onBarcodeResult(AccessoryBarcodeResult result) {
      if (result.status == NurApiErrors.NO_TAG) {
      // no barcode found
         txtStatus="No barcode found";
         isScanning=false;
      }
      else if (result.status == NurApiErrors.NOT_READY) {
      // cancelled
         txtStatus = "Cancelled";
      }
      else if (result.status == NurApiErrors.HW_MISMATCH) {
      // device does not support imager
         txtStatus = "No hardware found";
         isScanning=false;
      }
      else if (result.status != NurApiErrors.NUR_SUCCESS) {
      // error
         txtStatus = "Error: " + result.status;
         isScanning=false;
      }
      else { // Barcode scan success. Show result on the screen
         txtStatus = result.strBarcode;

         try {
            mAccessoryApi.beepAsync(100); //Beep on device
         }
         catch (Exception ex) { }

         isScanning=false;
      }
      updateUI();
   }
};
```

Handle device I/O events just like it has been done for the tag inventories

```
private NurApiListener mNurApiEventListener = new NurApiListener()
{
   ...
   @Override
   public void IOChangeEvent(NurEventIOChange event) {
      HandleIOEvent(event);
   }
   ...
};

private void HandleIOEvent(NurEventIOChange event)
{
   try {
      switch (event.source) {
         case 100: // trigger down
            if(event.direction == 1) {
               if(isScanning) // scan is in progress and we need to abort it
                   mAccessoryApi.cancelBarcodeAsync();
               else {
                  isAiming = true;
                  mAccessoryApi.imagerAIM(isAiming);
                  txtStatus = "Aiming...";
               }
            }
            else { // trigger released
               if(isScanning) { isScanning=false; return; }

            // Trigger released. Stop Aiming and start Scanning
               mAccessoryApi.imagerAIM(isAiming);
               mAccessoryApi.readBarcodeAsync(5000); // 5 sec timeout
               isScanning = true;
               txtStatus = "Scanning...";
            }
            break;
         default:
            break;
      }
   }
   catch (Exception ex) { txtStatus = ex.getMessage(); }

   updateUI();
}
```

3 Developing iOS applications

3.1 Install Xcode

Xcode version 9 or later is recommended in order to develop iOS applications communicating with the Alien ALR-S350 handheld:

https://developer.apple.com/xcode/

Below is a high level overview of the API framework. For more details on the API usage refer to the source code of a fully featured iOS demo application available at:

https://github.com/NordicID/nur_sample_ios

3.2 Using the API framework in your application

3.2.1 NurAPI Bluetooth framework

The framework provides an interface that works as a bridge between the NurAPI and the iOS Bluetooth stack. It is mainly responsible for providing a mechanism for NurAPI to communicate with the `CoreBluetooth` framework in iOS as well as relaying events from NurAPI to the application. It also provides a simpler API to scan for and connect to RFID devices.

The `NurAPIBluetooth` framework is accessible from both Objective-C and Swift.

3.2.2 Using the framework from Objective-C

Drag the framework into an application. Add the framework to *Embedded binaries* section in the *General* tab of your app target. Now you can import the `<NurAPIBluetooth/Bluetooth.h>` header:

```
#import <NurAPIBluetooth/Bluetooth.h>
```

All functionality is accessed through a singleton method, for example:

```
[[Bluetooth sharedinstance] startScanning];
```

All results from the `Bluetooth` class are delivered to registered delegates. An application that uses the class should have some component implement the `BluetoothDelegate` and register it as a delegate, for example:

```
// a class that implements the BluetoothDelegate protocol
@interface SelectReaderViewController : UIViewController <BluetoothDelegate>
...
@end

@implementation SelectReaderViewController
...

- (void)viewDidLoad {
    [super viewDidLoad];

    // register as a delegate
    [[Bluetooth sharedInstance] registerDelegate:self];
    ...
}

@end
```

See the `BluetoothDelegate` for all the methods that can be implemented.

When a connection to a reader is formed all communication with the device is performed through the low level NurAPI functions. These require a handle which can be accessed from the `nurapiHandle` property like this:

```
// start an inventory stream
int error = NurApiStartInventoryStream( [Bluetooth sharedInstance].nurapiHandle,
                                        rounds, q, session );
if ( error != NUR_NO_ERROR ) {
    // failed to start stream
}
```

Refer to the header for more detailed instructions on how to access the available functionality.

3.2.3 Thread model

All callbacks to the `BluetoothDelegate` are on different threads than the main application thread. This means that care needs to be taken when accessing application data and the UI. All low level NurAPI calls should also be performed on a secondary thread as some of the calls can block for a long time and deadlocks can occur if delegate methods are called. An example that fetches the NurAPI version string:

```
// a queue used to dispatch all NurAPI calls
@property (nonatomic, strong) dispatch_queue_t dispatchQueue;
...

// use the global queue with default priority
self.dispatchQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
...


dispatch_async(self.dispatchQueue, ^{
   char buffer[256];
   if ( NurApiGetFileVersion( buffer, 256 ) ) {
       NSString * versionString =
                 [NSString stringWithCString:buffer encoding:NSUTF8StringEncoding];

       // set the UILabel on the main thread
       dispatch_async(dispatch_get_main_queue(), ^{
           self.nurApiVersionLabel.text =
                 [NSString stringWithFormat:@"NurAPI version: %@", versionString];
       } );
   }
   else {
       // failed to get version
       ...
   }
} );
```

3.2.4 Using the framework from Swift

Drag the framework into an application. Add the framework to *Embedded binaries* section in the *General* tab of your app target. In order to use the framework in Swift code the project needs a bridging header. The contents of the *bridging header* is simply:

```
#import <NurAPIBluetooth/Bluetooth.h>
```

Now the framework can be imported in Swift with:

```
import NurAPIBluetooth
```

Once the bridging header is in place the framework can be used just as in the Objective-C

3.2.5 Objective-C demo

This is a demo application that shows how to use NURAPI in an Objective-C application. To build you need the `NURAPIBluetooth` framework. Drag and drop it from Finder into the `Frameworks` group and then add the framework to *Embedded binaries* section in the General tab of your application target.

The application builds both for the iOS simulator and real devices, but the simulator does not have Bluetooth support. The application will start in the simulator, but the Bluetooth subsystem will simply never be enabled.

3.2.6 Swift demo

This is a minimal demo that shows how to use the NURAPI framework from a Swift application. Add the framework to the project as per the Objective-C version. There is a *bridging header* that makes the framework available to the Swift code.

Functionality wise the Swift version contains the same storyboard, but only the initial view controller that starts the scanning and lists the found readers is implemented.

3.2.7 Using the framework with multiple architectures

The `NURAPIBluetooth` framework contains shared libraries for both device and simulator architectures. This allows the same framework to be used for both simulator testing as well as deploying on devices. When an application is submitted to the App Store the i386 simulator libraries cannot be included and must be stripped away from the build. This can be easily done with a build phase script that is executed last in the build.

The relevant script is below. It strips out all architectures that are not currently needed. If your `Info.plist` is in a different location, adapt the path on the following line:
`FRAMEWORK_EXECUTABLE_NAME=$(defaults read "$FRAMEWORK/Info.plist" CFBundleExecutable).`

```
APP_PATH="${TARGET_BUILD_DIR}/${WRAPPER_NAME}"

# This script loops through the frameworks embedded in the application and
# removes unused architectures.
find "$APP_PATH" -name '*.framework' -type d | while read -r FRAMEWORK
do
    FRAMEWORK_EXECUTABLE_NAME=$(defaults read "$FRAMEWORK/Info.plist"
CFBundleExecutable)
    FRAMEWORK_EXECUTABLE_PATH="$FRAMEWORK/$FRAMEWORK_EXECUTABLE_NAME"
    echo "Executable is $FRAMEWORK_EXECUTABLE_PATH"

    EXTRACTED_ARCHS=()

    for ARCH in $ARCHS
    do
        echo "Extracting $ARCH from $FRAMEWORK_EXECUTABLE_NAME"
        lipo -extract "$ARCH" "$FRAMEWORK_EXECUTABLE_PATH" -o
"$FRAMEWORK_EXECUTABLE_PATH-$ARCH"
        EXTRACTED_ARCHS+=("$FRAMEWORK_EXECUTABLE_PATH-$ARCH")
    done

    echo "Merging extracted architectures: ${ARCHS}"
    lipo -o "$FRAMEWORK_EXECUTABLE_PATH-merged" -create "${EXTRACTED_ARCHS[@]}"
    rm "${EXTRACTED_ARCHS[@]}"

    echo "Replacing original executable with thinned version"
    rm "$FRAMEWORK_EXECUTABLE_PATH"
    mv "$FRAMEWORK_EXECUTABLE_PATH-merged" "$FRAMEWORK_EXECUTABLE_PATH"
done
```

For more information and the original script, see:

http://ikennd.ac/blog/2015/02/stripping-unwanted-architectures-from-dynamic-libraries-in-xcode/