Developer

# Qorvo Nearby Interaction iOS Developer Guide

Qorvo IoT App

Release E

# Contents:

# 1 Introduction

This document describes Qorvo Nearby Interaction iOS App version 1.3.5 and Qorvo NI Background version 1.1.2.

Nearby Interaction with Third Party Accessories was introduced starting from iOS 15 for iPhones equipped with a U1 or U2 chip:

- iPhone 11 (11, Pro and Pro Max)
- iPhone 12 (12, Mini, Pro and Pro Max)
- iPhone 13 (13, Mini, Pro and Pro Max)
- iPhone 14 (14, Plus, Pro and Pro Max)
- iPhone 15 (15, Plus, Pro and Pro Max)

With iOS 16 or later, new features were included to Nearby Interaction, like Camera Assistance, which is implemented in this project.

The Qorvo Nearby Interaction app is an XCode project written in Swift language. As a Nearby Interaction implementation, it has two communication layers: BLE and UWB. The BLE layer handles data transfer between iPhone and accessories, UWB is used, thereafter for FiRa Two Way Ranging.

# 2 QNI Architecture

The Main app is implemented in “QorvoDemoViewController.swift”, it uses BLE for discovery and exchange parameters between a Qorvo accessory and an iOS device, and then uses NI for FiRa compliant (version 1.3) Double Sided Two-Way Ranging (DS-TWR), using the parameters exchanged in the BLE phase.

The *NI communication flow* is shown in the following picture.

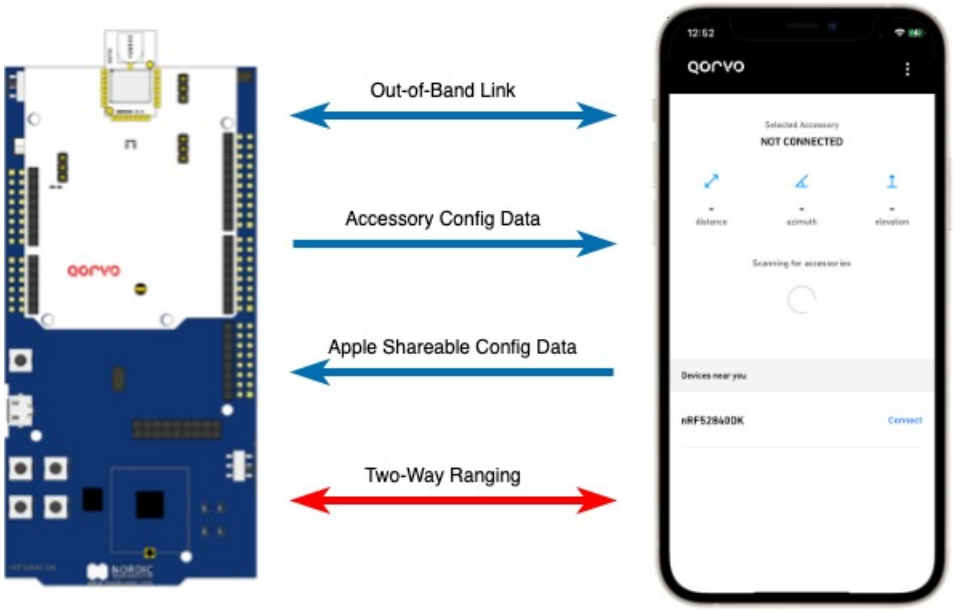


Fig. 2.1: Qorvo NI Communication Flow

**BLE overview**

BLE scans nearby devices based on their BLE services (Nordic Uart Service - NUS [QNI v2.0.5] or Qorvo NI Service - QNIS [QNI 3.0.0]), it makes a list of Qorvo NI accessories nearby. Using this list the main application shows devices to connect and has all the information required to communicate to that accessory.

**NI overview**

In the Main app, for each accessory in the Qorvo NI accessories list commanded to connect, a new NI Session will be created and associated to that accessory. Then when a start ranging command is sent to that accessory the associated NI Session starts, new distance and direction values will be received for that accessory and the correct fields will be updated based on the NISession.

**Camera Assistance**

When using iOS 16, Camera Assistance can be enabled for NI Sessions created, if not using Background mode. Camera Assistance can be used with both services (Nordic Uart Service - NUS [QNI v2.0.5] or Qorvo NI Service - QNIS [QNI 3.0.0]).

**Background Mode**

Starting on iOS 16, NI Sessions can be set to keep running when the app is sent to Background. Background NI Sessions use a different initialiser and have only distance information available, at a slower rate then regular NI Sessions. To keep the NI Session ranging when in Background the iOS app will check the Apple NI Service, which is only implemented in QNI 3.0.0.

## 2.1 Out of Band Messages Sequence

The project uses BLE as Out-of Band communication link (OOB), the BLE layer is based on the functions available by XCode Core Bluetooth API.

The iPhone will act as a Central (Scanner) connecting with the Qorvo accessory as Peripheral (Advertiser) using Qorvo NI Service (QNIS) or Nordic Uart Service (NUS).

Once the OOB link is established, both NI applications (iOS and Device) can exchange messages, always starting with the **SampleAppMessageId**.

The *NI OOB messages sequence* is shown in the following picture.

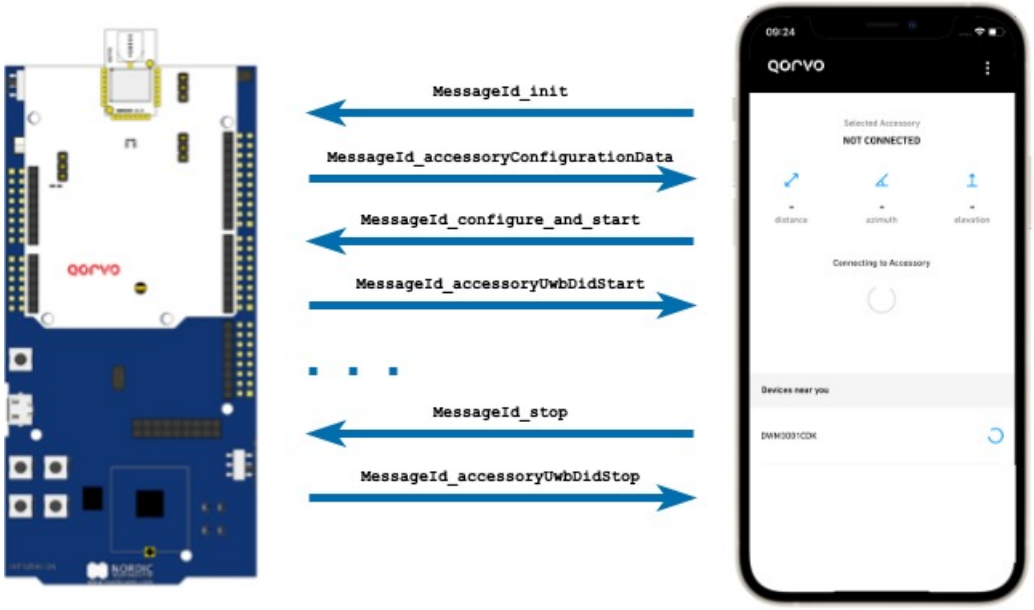


Fig. 2.2: NI OOB messages sequence

The message IDs are implemented in the `MessageId: UInt8` enum:

**initialize:** The first message is sent by the iOS device to Qorvo accessory. It is a single byte message with `MessageId.initialize`. After reception, the Qorvo accessory will use the NIQ library to generate the AccessoryConfigurationData and then, send it back to the iOS device via BLE.

**accessoryConfigurationData:** From the Qorvo accessory to the iOS device, the message starts with the `MessageId.accessoryConfigurationData` and has the Accessory Configuration Data as its payload.

**configureAndStart:** Next in the sequence, the iOS device generates and send the `shareableConfigurationData` structure, starting with `MessageId.configureAndStart`. The Qorvo accessory identify the packet and parse the payload using the NIQ Library (`niq_configure_and_start_uwb()`), after that it sends the `MessageId.accessoryUwbDidStart` message before switch to UWB and start Two-Way Ranging.

**accessoryUwbDidStart:** Last message from the Qorvo accessory to the iOS device before start Ranging. Single byte `MessageId.accessoryUwbDidStart`.

**stop:** From iOS device to the Qorvo accessory, single byte message requesting the accessory to stop ranging.

**accessoryUwbDidStop:** Response to `MessageId.stop` from the Qorvo accessory to the iOS device, single byte message confirming that the accessory has stopped.

Besides the OOB message sequence BLE has no much use in this project, and more Sample Message IDs can be created to interact with the Qorvo accessory.
In this iOS app version three new Message IDs were added:

**getReserved** Request from iOS device to the Qorvo accessory to send the current Device Struct, which will have device parameters and status. For future implementation.

**setReserved** Message from iOS device to the Qorvo accessory to set the Device Struct with the message payload data. For future implementation.

**iOSNotify** Message followed by a string, which will trigger a notification on the iOS device. Notifications only work when the app is in background.

## 2.2 BLE Layer

The BLE Layer is implemented in “BluetoothLECentral.swift”, it has all functions to support configuring the iOS device as a BLE Central device, send and receive data to/from the Qorvo accessory.

```
root
└── NINearbyAccessorySample            Project/sources folder
    └── BluetoothSupport               BLE folder
        └── BluetoothLECentral.swift   BLE Central implementation
```

The BLE class is:

```
class DataCommunicationChannel: NSObject {
   ...
}
```

It is instantiated in `QorvoDemoViewController` as:

```
var dataChannel = DataCommunicationChannel()
```

Then, the callback functions are assigned for BLE events:

```
dataChannel.accessoryDiscoveryHandler = accessoryInclude
dataChannel.accessoryTimeoutHandler = accessoryRemove
dataChannel.accessoryConnectedHandler = accessoryConnected
dataChannel.accessoryDisconnectedHandler = accessoryDisconnected
dataChannel.accessoryDataHandler = accessorySharedData
```

The main variable, shared across other classes, is the `qorvoDevices` array:

```
var qorvoDevices = [qorvoDevice]()
```

It is an array of class `qorvoDevice`, also declared in “BluetoothLECentral.swift”:

```
class qorvoDevice {
   var blePeripheral: CBPeripheral
   var scCharacteristic: CBCharacteristic
   var rxCharacteristic: CBCharacteristic
   var txCharacteristic: CBCharacteristic

   var bleUniqueID: Int
   var blePeripheralName: String
   var blePeripheralStatus: String
   var bleTimestamp: Int64
   var uwbLocation: Location

   init(...) {
      ...
   }
}
```

The qorvoDevice class is composed by the following variables:

**blePeripheral: CBPeripheral** The BLE Peripheral instance, used to handle the right peripheral, when requested.

**scCharacteristic: CBCharacteristic** Secure Characteristic, used for pairing, only required for Background mode.

**rxCharacteristic: CBCharacteristic** BLE Characteristic to be used by the accessory to receive data. The iOS device use it to send data to the Qorvo accessory.

**txCharacteristic: CBCharacteristic** BLE Characteristic to be used by the accessory to send data. On the iOS device, notification is enabled for this characteristic during the BLE connection step, and new messages from the Qorvo accessory will be received by a `peripheral didUpdateValueFor` event.

**bleUniqueID: Int** It is set with the `hashValue` parameter value, from the BLE device instance, assigned inside the `centralManager didDiscover` event. It is used to associate the BLE device to other project components.

**blePeripheralName: String** BLE device's name from the advertising packets, is assigned in the `centralManager didDiscover` event.

**blePeripheralStatus: String** BLE device status (statusDiscovered, statusConnected or statusRanging).

**bleTimestamp: Int64** Last time that the device adverstised, is used to apply a timeout and remove devices from the `qorvoDevices` array if not communicating for a long period.

**uwbLocation: Location** Last positioning information available:

```
struct Location {
   var distance: Float
   var direction: simd_float3
   var elevation: Int
   var noUpdate: Bool
}
```

The Location structure is declared in "BluetoothLECentral.swift":

**distance: Float** Last distance reported, in meters.

**direction: simd_float3** Last direction vector reported.

**elevation: Int** When direction is not available, elevation is estimated from *Convergence State*.

**noUpdate: Bool** When direction is not available this flag is set to "True" and user will know that information is not updated.

BLE is started by calling `dataChannel.start()`, and the iOS BLE will start scan.

The BLE events are handled by `centralManager` and `peripheral` functions, and in the same function the associated callback is checked and executed, the events are:

**didDiscover** Happens if a new advertising packet is received, it checks if the peripheral is already in the `qorvoDevices` array, using the `peripheral.hashValue`. If not included, it is appended to the array and calls `accessoryDiscoveryHandler()`, if already included it only updates the `qorvoDevice.bleTimestamp`.

**didFailToConnect** Reports a failed try to connect, it calls a placeholder function `cleanup()` which is not implemented in this app.

**didConnect** After a successful connection it calls `peripheral.discoverServices()`, to parse the BLE Services, and then its characteristics.

**didDisconnectPeripheral** Happens when the peripheral disconnects, set the `qorvoDevice.blePeripheralStatus` to `statusDiscovered` and update the `qorvoDevice.bleTimestamp` to avoid an early timeout. Finally it calls the `accessoryDisconnectedHandler()`, there is also a logic placeholder to retrieve the peripheral connection, but the embedded app default action is to stop an ongoing NI Session, disconnect BLE and start to advertise again. If the user needs the application to retrieve a connection, the retrieval algorithm needs to be planned and implemented on both sides (iOS and embedded).

Following the `centralManager` events (more related to connection) it starts the `peripheral` events:

**didModifyServices** Reacts to peripheral services invalidation. The default action is to run `peripheral.discoverServices()` again.

**didDiscoverServices** Result from a previous `peripheral.discoverServices()` action, the next step is to discover the peripheral characteristics, by calling `peripheral.discoverCharacteristics()`

**didDiscoverCharacteristicsFor** When a characteristic is discovered it is parsed by its UUID and assigned to the related parameter in `qorvoDevice`. Also it sets notification for Tx characteristics (txCharacteristic will send data from the Qorvo accessory to the iOS device), and calls the `accessoryConnectedHandler()` callback.

**didUpdateValueFor** Reacts to data arrival through the characteristic notification. The `accessoryDataHandler()` callback is called.

**didUpdateNotificationStateFor** Result when a characteristic is requested to notify.

Other functions part of `DataCommunicationChannel`:

```
func start() {
    ...
}
```

To be called after instantiation and `dataChannel` callbacks assignment, it will demand the iOS device to start scanning using `TransferService` and `QorvoNIService` UUIDs as filter.

```
func connectPeripheral(_ uniqueID: Int) throws {
    ...
}
```

Request BLE to connect to an accessory, identified by its bleUniqueID. If the device is in the `qorvoDevices` array and the `qorvoDevice.blePeripheralStatus` is `statusDiscovered` it will request a connection using `centralManager.connect()`.

```
func disconnectPeripheral(_ uniqueID: Int) throws {
    ...
}
```

Request BLE to disconnect from an accessory, identified by its bleUniqueID. The function tests if the device is in the `qorvoDevices` array and the `qorvoDevice.blePeripheralStatus` is not `statusDiscovered` (already disconnected).

```
func sendData(_ data: Data,_ uniqueID: Int) throws {
    ...
}
```

Send a Data pack to the accessory, identified by its bleUniqueID. Before sending it checks if the device is in the `qorvoDevices` array.

```
@objc func timerHandler() {
    ...
}
```

Timer Handler for timeout management, time interval is set to 0.2 seconds, the function checks the `qorvoDevices` array for devices which `qorvoDevice.blePeripheralStatus` is `statusDiscovered` and the last `qorvoDevice.bleTimestamp` happened more than 5 seconds before. If the condition matches, the device is removed from the `qorvoDevices` array.

```
func getDeviceFromUniqueID(_ uniqueID: Int)->qorvoDevice {
    ...
}
```

Suporting function, that return the device instance from the `qorvoDevices` array that matches the bleUniqueID.

## 2.3 NI Layer

The NI Layer is part of the main app, and implemented in “QorvoDemoViewController.swift”, it has all functions to support configuring the iOS device as a BLE Central device, send and receive data to/from the Qorvo accessory.

```
root
└── NINearbyAccessorySample            Project/sources folder
    └── QorvoDemoViewController.swift  Main app
```

To connect to multiple accessories, one NI Session is created for each accessory connected.
NI Sessions are associated to accessories using the `bleUniqueID: Int` in a Dictionary:

```
var referenceDict = [Int:NISession]()
```

After a Qorvo accessory is connected, the BLE connection handler calls the funtion set for `accessoryConnectedHandler()`, which is assigned to `accessoryConnected` in “QorvoDemoViewController.swift”.
In the `accessoryConnected` a new NI Session is created for the new device.

```
// Create a NISession for the new device
referenceDict[deviceID] = NISession()
referenceDict[deviceID].delegate = self
referenceDict[deviceID].setARSession(arView.session)
```

The Qorvo Nearby Interaction app enables Camera Assistance for all NI Sessions (not available in Background mode version). If using a single Qorvo device the only configuration required is to set the configuration parameter `.isCameraAssistanceEnabled` to `true`, for multiple devices an AR Session needs to be previously declared (*Auxiliary AR Session*), initialised with the right parameters and assigned to each NI Session, by using `.setARSession()`.
The new NI Session is started when a `MessageId.accessoryConfigurationData` message is received, the `accessoryDataHandler` is set to `accessorySharedData()`, which parse the BLE packet and call `setupAccessory()`.
This function set the `NINearbyAccessoryConfiguration` value and starts the NI Session using that configuration:

```
configuration = NINearbyAccessoryConfiguration(data: configData)
configuration.isCameraAssistanceEnabled = true
referenceDict[deviceID].run(configuration)
```

For a Background enabled NI Session, the initialisation is slightly different:

```
let peerDevice = dataChannel.getDeviceFromUniqueID(deviceID)
let peerIdentifier = peerDevice.blePeripheral.identifier

configuration = NINearbyAccessoryConfiguration(accessoryData: configData,
                                               bluetoothPeerIdentifier: peerIdentifier)
referenceDict[deviceID].run(configuration)
```

Background NI Sessions are initialised with a second parameter `bluetoothPeerIdentifier`, which is recovered from the BLE device instantiation (first two lines).
Camera Assistance doesn't work for Background NI Sessions and set `.isCameraAssistanceEnabled` to true takes no effect.

### 2.3.1 NISessionDelegate

Functions to handle NI Sessions event are implemented in a `QorvoDemoViewController` extension, NISessionDelegate:

**didGenerateShareableConfigurationData:** Happens after start the NI Session using the command `.run(configuration)`. The `shareableConfigurationData` for that NI Session was generated and needs to be sent to the Qorvo Accessory via BLE. The `shareableConfigurationData` is the payload of a `MessageId.configureAndStart` message, by now the iOS device is ready to start the Two-Way Ranging rounds.

**didUpdateAlgorithmConvergence:** Update the `NIAlgorithmConvergence` status. `.horizontalAngle` and `.verticalDirectionEstimate` can be used when direction is not available, the `isConverged` flag set here will define if Converged data is ready. It can also guide the user on how to proceed to achieve Convergence (environment light, motion).

**didUpdate:** After a successful Ranging round, a `didUpdate` event will happen and updated distance and direction (see *iPhone Line of Sight*) info will be available.

**didRemove:** Event when a NI Session timed out, the `shouldRetry()` function checks if the device is still connected to BLE, send a `MessageId.stop` and then `MessageId.initialize`. Retrieval logic can be added or modified for the application purpose.

**didInvalidateWith:** Happens when the NI Session fails to start, then it parses the `NIError` and report the issue.

**sessionWasSuspended():** Called when the NI Session is suspended.

**sessionSuspensionEnded():** Called when the NI Session is resumed.

## 2.4 Camera Assistance

Camera Assistance enhances Nearby Interaction by leveraging the device trajectory computed from ARKit. It associates an AR Session to a NI Session, making distance and direction more consistently available than using Nearby Interaction alone.

“An iPhone detects a peer device’s direction when it appears within the narrow line of sight illustrated by the conic region in the following diagram.”

### 2.4.1 iPhone Line of Sight

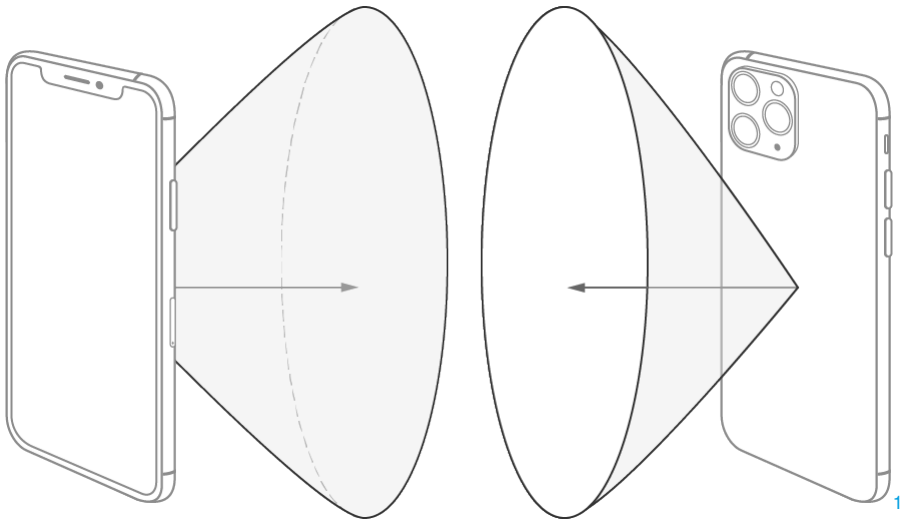

Fig. 2.3: Coach the user on range, orientation, and line of sight

When out of the line of sight, the direction updates would stop and only distance would be available. Camera Assistance effectively broads the UWB field of view, making direction available when out of the line of sight.

[1] https://developer.apple.com/documentation/nearbyinteraction/initiating_and_maintaining_a_session#3608165

### 2.4.2 Auxiliary AR Session

As mentioned in *NI Layer*, to enable Camera Assistance for a single NI Session the only configuration required is to set the configuration parameter `.isCameraAssistanceEnabled` to `true`. An AR Session will be automatically created whithin NI Interaction framework. When the NI Session is set to run, this AR Session will be set to run too.

```
configuration = NINearbyAccessoryConfiguration(data: configData)
configuration.isCameraAssistanceEnabled = true
referenceDict[deviceID].run(configuration)
```

To enable Camera Assistance for multiple devices an AR Session needs to be previously declared and initialised with the following parameters:

```
@IBOutlet weak var arView: ARView!
let arConfig = ARWorldTrackingConfiguration()

...

override func viewDidLoad() {
   super.viewDidLoad()
   ...

   // Set AR Session configuration to provide camera assistance to new NI Sessions
   arConfig.worldAlignment = .gravity
   arConfig.isCollaborationEnabled = false
   arConfig.userFaceTrackingEnabled = false
   arConfig.initialWorldMap = nil

   // Start the AR Session with the previous set configuration
   arView.session = ARSession()
   arView.session.delegate = self
   arView.session.run(arConfig)
   arView.scene.addAnchor(anchor)

   ...
}
```

Then, this AR Session will be assigned to each NI Session, by using `.setARSession()`:

```
// Create a NISession for the new device
referenceDict[deviceID] = NISession()
referenceDict[deviceID].delegate = self
referenceDict[deviceID].setARSession(arView.session)
```

Still, the configuration parameter `.isCameraAssistanceEnabled` needs to be set to `true`.

---

**Note:** If only defining `.isCameraAssistanceEnabled` as `true` for multiple accessories (not using `.setARSession()`) each new NI Session set to run will invalidate the previous NI Session.

---

## 2.5 AR Session

The AR Session is not only used to enables Nearby Interaction Camera Assistance. It also is used to plot 3D objects in each NI Session position reported. The *Auxiliary AR Session* was created inside an ARView extension named `WorldView`, this component is able to presents graphics rendered with Apple's RealityKit:

"The ARView class allows you to display a 3D interactive AR scene using the RealityKit framework. It lets you easily load, manipulate, and render 3D AR content in your application."

The AR Session is always running, even when worldView is hidden, and shouldn't be stopped.
To plot virtual objects in the WorldView, AR Entities are created and associated to accessories using their `bleUniqueID: Int` in a Dictionary:

```
var entityDict = [Int:ModelEntity]()
```

The `WorldView` component also implements the functions needed to handle the AR Entities.
`pinShape` and `material` are declared in `WorldView` to support new AR Entities creation, and can be changed by the user.
An Anchor Entity is also created to initialise the AR Session, shown in *Auxiliary AR Session*, the anchor position is set to the AR View origin (0, 0, 0), and will be used as reference for the new AR Entities.

```
let anchor = AnchorEntity(world: SIMD3(x: 0, y: 0, z: 0))
let pinShape = MeshResource.generateSphere(radius: 0.05)
let material = SimpleMaterial(color: .yellow, isMetallic: false)
```

The new entities are created and associated to the device inside the `accessoryConnected()` call, same function where the new NI Sessions are created:

```
// Also creates the AR object
worldView.insertEntity(deviceID)
```

In `WorldView`, the new Model Entity is created using `pinShape` and `material` previously declared, then the entity position is set to a high value, to place it out of the field of view[2]. Finally the new entity is added to the anchor.

```
func insertEntity(_ deviceID: Int) {
   // Create a new entity include ie to the anchor
   entityDict[deviceID] = ModelEntity(mesh: pinShape, materials: [material])
   entityDict[deviceID]!.position = [0, 0, 100]
   anchor.addChild(entityDict[deviceID]!)
}
```

[2] When created, the entity will be static, waiting for the next position update. If placed on the origin it may take a short time until it updates to the right position.

When the associated device has an update (`NISessionDelegate::didUpdate()`), the AR Entity position is updated too if `worldView` is not hidden.

```
// Update AR anchor
if !worldView.isHidden {
   guard let transform = session.worldTransform(for: accessory) else {return}
   worldView.updateEntityPosition(deviceID, transform)
}
```

## 2.6 GUI Assets

All the updates from the connected Qorvo devices are shown using both text and image components, to organise these components a vertical Stack View (`mainStackView`) is used. The components placed in the `mainStackView` are (in order):

### 2.6.1 QorvoAccessorySample/WorldView/WorldView.swift

WorldView is an ARView (*AR Session*), instantiated in `QorvoDemoViewController` as worldView:

```
let worldView = WorldView(frame: .zero)
```

It is initialy hidden in the Qorvo Nearby Interaction app. It is expanded by pressing the `arButton`, that will call the `SwitchAR()` function:

```
UIView.animate(withDuration: 0.4) {
   self.worldView.isHidden = false

   self.deviceView.isHidden = true
   self.locationFields.isHidden = true
   self.arrowView.isHidden = true
   self.accessoriesTable.isHidden = true
}
```

To free screen space, the other components are hidden, except for a UI View `SeparatorView`. The `SwitchAR()` function toggles the WorldView "`.isHidden`" property, after checking its current state.

### 2.6.2 QorvoAccessorySample/DeviceView/DeviceView.swift

UI View with two Labels, instantiated in `QorvoDemoViewController` as deviceView:

```
let deviceView = DeviceView()
```

The label `deviceName` show the name of the current selected device. Each device "Mini-Field" has a button that allow the user to select that device, when pressed, the button calls `selectDevice()` where one of the actions is to call `.setDeviceName()` with the `.blePeripheralName` or `"NOT CONNECTED"`.

### 2.6.3 QorvoAccessorySample/LocationFields/LocationFields.swift

A UI View with a sequence of three sets of icons and labels to exhibit distance, azimuth and elevation.

```
let locationFields = LocationFields()
```

The base component, called `Field`, is composed by an Image View (for icon) and Text Fields (for title and values). `LocationFields` instantiate three `Field` changing their icons and titles, and implement the functions to update their values of distance and direction, which are called periodically by the `NISessionDelegate::didUpdate()` function.

```
let distance  = Field(image: UIImage(named: "distance_icon"),  fieldTitle: "distance")
let azimuth   = Field(image: UIImage(named: "azimuth_icon"),   fieldTitle: "azimuth")
let elevation = Field(image: UIImage(named: "elevation_icon"), fieldTitle: "elevation")
```

### 2.6.4 QorvoAccessorySample/ArrowView/ArrowView.swift

UI View `ArrowView` is composed by a SceneKit View `arrowImgView`, an animated Image View `scanning`, and a Label `infoLabel`.

```
let arrowView = ArrowView()
```

The `arrowImgView` loads a `scene` based on a 3D model "3d_arrow.usdz", this model is used for both 2D and 3D Arrows. When in 3D the scene light is on, and the root node light color is `UIColor.darkGray`, this way the arrow can be seen against the white color and the shadow casted show its shape, and where it is pointing. The 3D model is turned in the X axis (Elevation), Y axis (Azimuth), and the Z axis is always zero. When in 2D mode, the light scene is off and the root node light color is `UIColor.black`, this way it gives the model a flat apearence. The 3D model is turned only in the Z axis using the Azimuth angle, X axis and Z axis are constants (X = 90° and Y = 0°).

ArrowView implements the functions:

**initArrowPosition():** Simple set the arrow orientation to point it forward.

**switch3DArrow():** Set light and color of the 3D model to give it a flat apearence when the arrow is set to 2D, or revert back to 3D.

**setArrowAngle():** Check if the arrow is on 3D or 2D mode and calculate the angles to turn the 3D arrow to the given elevation and azimuth information. If in 2D mode will turn only in the 2D screen plane, like a compass.

**setScanning():** The image View `scanning` shows an indicator that the iPhone is scanning for Qorvo devices. To create an animation effect an `UIImage` with the scanning indicator "spinner.svg" is declared, turned and loaded to an `UIImage` array `imageScanning`, this array is set to `scanning.animationImages` to use the `UIImageView` animation feature:

```
let scanning = UIImageView()
var imageScanning = [UIImage]()

...

 override init(frame: CGRect) {
   // Add subviews to the parent view
   super.init(frame: frame)
   ...

   // Prepare the "scanning" animation
   let image = UIImage(named: "spinner.svg")!
   for i in 0...24 {
      imageScanning.append(image.rotate(radians: Float(i) * .pi / 12)!)
   }
```

(continues on next page)

(continued from previous page)

```
    scanning.animationImages = imageScanning
    scanning.translatesAutoresizingMaskIntoConstraints = false
    scanning.animationDuration = 1
    scanning.isHidden = false
    scanning.startAnimating()
    ...
}
```

After that the image View `scanning` can be controlled using `scanning.startAnimating()` or `scanning.stopAnimating()`, and hidden/shown using the parameter `scanning.isHidden`, that are set when calling `.setScanning`.

**infoLabelUpdate():** The Label `infoLabel` report events to the user. The text can be set using this function.

### 2.6.5 QorvoAccessorySample/SeparatorView/SeparatorView.swift

A simple light gray UI View with a fixed Label “Devices near you”, instantiated in QorvoDemoViewController:

```
let separatorView = SeparatorView(fieldTitle: "Devices near you")
```

It separates the main views (WorldView and ArrowView) from the device list. In QorvoDemoViewController, gestures are associated to the `separatorView`: upSwipe and downSwipe.

```
override func viewDidLoad() {
    super.viewDidLoad()
    ...

    // Add gesture recognition to "Devices near you" UIView
    let upSwipe   = UISwipeGestureRecognizer(target: self, action: #selector(swipeHandler))
    let downSwipe = UISwipeGestureRecognizer(target: self, action: #selector(swipeHandler))

    upSwipe.direction   = .up
    downSwipe.direction = .down

    separatorView.addGestureRecognizer(upSwipe)
    separatorView.addGestureRecognizer(downSwipe)
}
```

When a gesture is recognised it is called the `swipeHandler()` function, if the gesture direction is “up” it will hide `locationView` if in regular mode, or show the `accessoriesTable` if in AR mode. If the gesture direction is “down” it will show `locationView` if in regular mode, or hide the `accessoriesTable` if in AR mode.

### 2.6.6 QorvoAccessorySample/AccessoriesTable/AccessoriesTable.swift

This is the most complex component, it implements Delegate in the `QorvoDemoViewController` to handle the single cells, each cell has gestures, buttons and a mix of text and graphics implemented in “AccessoriesTable.swift”.

```
let accessoriesTable = AccessoriesTable()
```

Each new Qorvo device found includes a new cell, implemented in SingleCell.swift, to the `accessoriesTable`. The accessory can be handled and updated TWR info can be seen in the device cell. The cell order can change anytime, when devices are added/removed (due timeout) to/from `qorvoDevices`, so the cell has one parameter to define to which device that cell is related: `cell.uniqueID` and `cell.accessoryButton.tag`, set inside `UITableViewDelegate::cellForRowAt()`, the related `qorvoDevice` from `qorvoDevices` can be selected using `.getDeviceFromUniqueID()`.


Fig. 2.4: Relation between qorvoDevices array and accessoriesTable.

The SingleCell components are:

**accessoryButton:** Button to select the accessory, it covers the entire cell and its title is the device name. When a new cell is created its `accessoryButton.tag` is set with the same `cell.tag` value, and the target action is to call the `buttonSelect()` function. `buttonSelect()` will check the `accessoryButton.tag` to get the related `qorvoDevice` from `qorvoDevices`.

```
@IBAction func buttonSelect(_ sender: UIButton) {

   if dataChannel.getDeviceFromUniqueID(sender.tag) != nil {
      let deviceID = sender.tag

      selectDevice(deviceID)
      logger.info("Select Button pressed for device \(deviceID)")
   }
}
```

**actionButton:** Button to request the BLE connection, it is visible when the device is added and hidden when the device is Ranging. Using the same logic as the `accessoryButton` the `actionButton.tag` is set to make possible to identify the device requested. Its target action is to call the `buttonAction()` function, which request the connection by calling `connectPeripheral()`.

```
@IBAction func buttonAction(_ sender: UIButton) {
   let deviceID = sender.tag

   if let qorvoDevice = dataChannel.getDeviceFromUniqueID(deviceID) {

      // Connect to the accessory
      if qorvoDevice.blePeripheralStatus == statusDiscovered {
         arrowView.infoLabelUpdate(with: "ConnectingAccessory".localized)
         connectToAccessory(deviceID)
      }
      else {
         return
      }
```

(continues on next page)

(continued from previous page)

```
        // Edit cell for this sender
        accessoriesTable.setCellAsset(deviceID, .loading)

        logger.info("Action Button pressed for device \(deviceID)")
    }
}
```

**miniLocation:** A View for the components that show Distance (UI Label `distanceLabel`) and Azimuth (UI Label `azimuthLabel` and `miniArrow`) information. The Labels are set directly using the `.text` parameter, the `miniArrow` is turned using the `.transform` parameter, in the `updateMiniFields()` function.

**loading:** Image to show that the device connection is in progress. Created the same way as the `scanning` in `arrowView` but using "spinner_small.svg" as base image.

The assets for each cell in `accessoriesTable` are shown/hidden based on the device state. The `DeviceTableViewCell` class has a function, `selectAsset()`, to set them:

`.actionButton`: When the cell is created, only the `actionButton` "Connect" is shown besides the device name;

`.loading`: After clicking on "Connect" the loading animation is shown in the same position where it was the "Connect" button.

`.miniLocation`: When TWR starts the `miniLocation` View, where the TWR info will be updated, is shown.

Also, gestures are enabled for each cell in `AccessoriesTable` by including the function `trailingSwipeActionsConfigurationForRowAt`. There, a `UIContextualAction disconnect` is created, and associated to the `swipeActions`. The `disconnect` action sends `MessageId.stop` to the associated Qorvo device.

# 3 Important Notice for iPhone 14 and newer models

The NISession may not offer the same device capabilities on the iPhone 14 and newer lineups as it does on previous iPhone models, resulting in varying behaviors and feature limitations.

**The primary distinctions are:**

- To obtain AoA measurements, NISession must be in a converged state.
- The AoA elevation is not represented as an angle, but instead as an estimated enumeration value.

## 3.1 NISession.DeviceCapabilities Differences

| DeviceCapabilities | Previous iPhone | iPhone 14 and newer |
|---|---|---|
| supportsPreciseDistanceMeasurement | true | true |
| supportsDirectionMeasurement | true | false |
| supportsCameraAssistance | true | true |
| supportsCoarseDistanceMeasurement | true | false |
| supportsFineRanging | true | false |
| supportsCoarseRanging | true | false |
| supportsAoA | true | false |
| supportsSyntheticAperture | true | true |

## 3.2 Impact on features / App behaviors

| DeviceCapabilities | Previous iPhone | iPhone 14 and newer |
|---|---|---|
| AR | enabled | disabled |
| 3D Arrow | enabled | disabled |
| Elevation Angle | enabled | partially disabled - only estimation (see Convergence chapter) |

## 3.3 Convergence State

Once the UWB session has started, the application will guide you through the process of achieving a converged state. (Movement in the space, More light required,..). The convergence is obtained using the iPhone's sensors (mainly camera sensor).

To verify the convergence value, you can use the following function implementation:

```
func session(_ session: NISession,
             didUpdateAlgorithmConvergence convergence: NIAlgorithmConvergence,
             for object: NINearbyObject?) {
```

(continues on next page)

(continued from previous page)

```
    // Test if NISession converged
}
```

**Following convergence, the didUpdate function of the NISession delegate will provide the following information:**

- horizontalAngle: The AoA azimuth in radians.
- verticalDirectionEstimate: An enumerated value that estimates the vertical position of the nearby object, which can be one of the following: ABOVE, BELOW, SAME, ABOVEORBELOW, or UNKNOWN.

# 4 Required Tools

To test, debug or implement your own Nearby Interaction Aplication for iOS it will be required an Apple Developer Account.

It can be created for free at:

Apple Developer[3]

## 4.1 Software

To test the Qorvo Nearby Interaction App, first download the Xcode from Apple App Store:

Apple App Store - Xcode[4]

**Warning:** Xcode is available only for macOS devices.

## 4.2 Hardware

The app will be deployed to real hardware, so it is required an iPhone equipped with a U1 or U2 chip:

- iPhone 11 (11, Pro and Pro Max)
- iPhone 12 (12, Mini, Pro and Pro Max)
- iPhone 13 (13, Mini, Pro and Pro Max)
- iPhone 14 (14, Plus, Pro and Pro Max)
- iPhone 15 (15, Plus, Pro and Pro Max)

**Warning:** The iPhone must be updated to the iOS 15 or later.

[3] https://developer.apple.com/
[4] https://apps.apple.com/us/app/xcode/id497799835

# 5 Building/Deploying the iOS app

Open Xcode, click on File→Open. . . , navigate to the Qorvo NI Project folder and open the NINearbyAccessorySample.xcodeproj file.

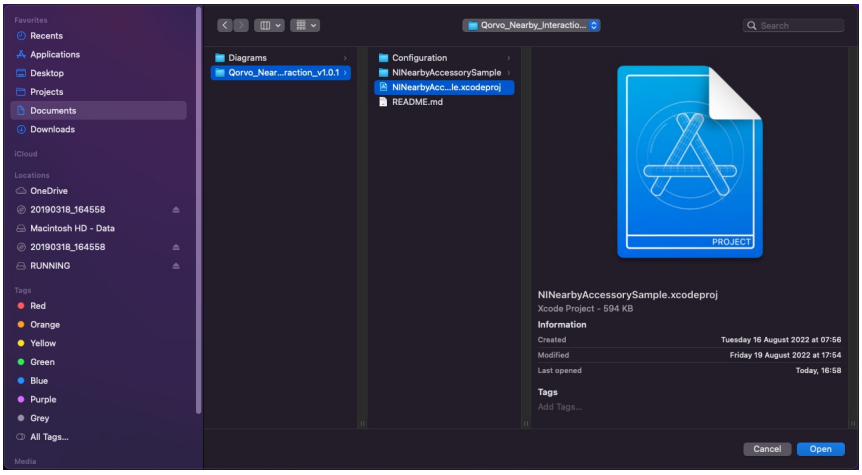


Fig. 5.1: Open the Sample Project in Xcode

In the menu on the left, select the project file (click on the project name, NINearbyAccessorySample), then click on the “Signing & Capabilities” tab and set value for Team.

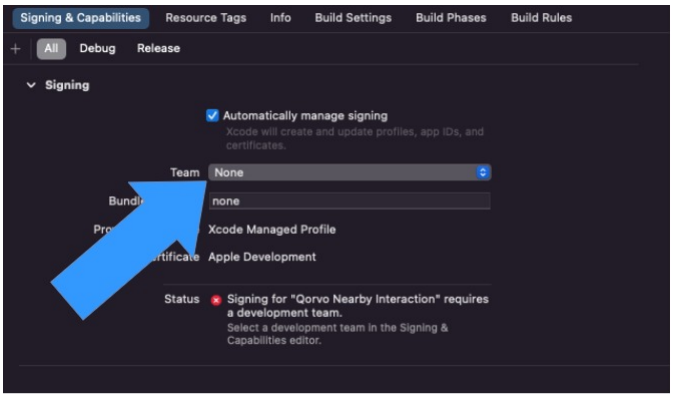


Fig. 5.2: Select the Team to be able to build

Connect the target iPhone to a free USB port of the macOS device. Select the device (iPhone) on the top navigation bar (*Change the default iOS device to the connected iPhone model*) and click on the play icon button on the left

(*Build/Start Application in the connected device*). The app will be built and deployed to the iPhone.

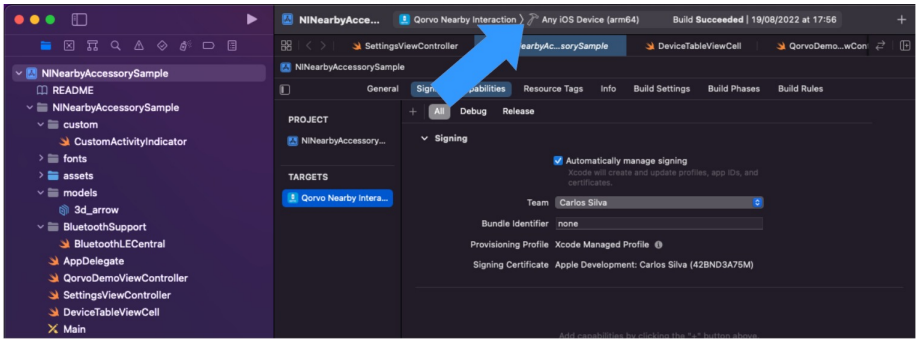


Fig. 5.3: Change the default iOS device to the connected iPhone model

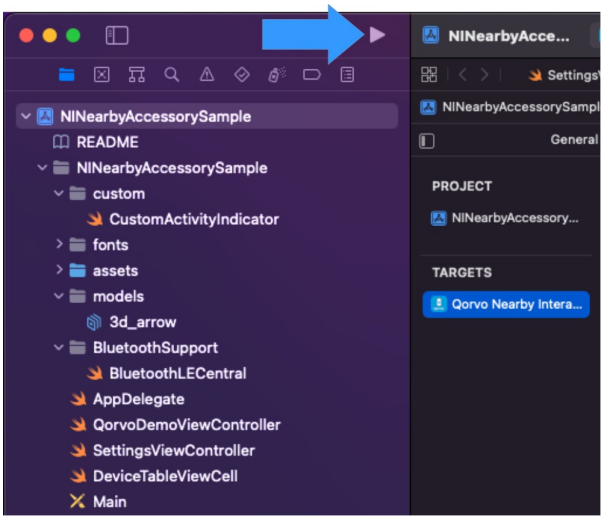


Fig. 5.4: Build/Start Application in the connected device

When the Qorvo Nearby Interaction application opens it starts to scan for nearby accessories (*Qorvo Nearby Interaction application scanning for Qorvo devices*), if a Qorvo device is found it is added to the “Devices near you” list.

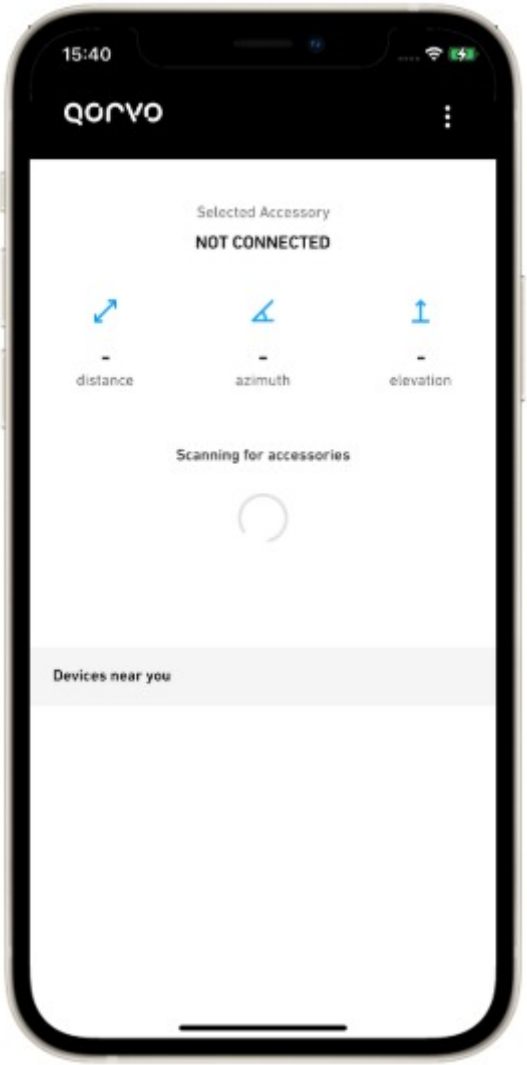


Fig. 5.5: Qorvo Nearby Interaction application scanning for Qorvo devices

For information on Qorvo Kits compatible with Nearby Interaction check:

UWB SOLUTIONS COMPATIBLE WITH APPLE'S U1 CHIP[5]

[5] https://www.qorvo.com/innovation/ultra-wideband/products/uwb-solutions-compatible-with-apple-u1

# 6 Building/Debugging the QNI Embedded Project

The QNI firmware is complementary to the Qorvo Nearby Interaction for iOS, so is useful being able to follow the device status when learning or debugging your own code.

## 6.1 Required Tools

The QNI sources include Segger Embedded Studio projects to each supported target. The sources are common to all ".emProject" projects.

### 6.1.1 Software

To debug or implement your own changes to the QNI application it will be required Segger Embedded Studio [version 5.x]:

Segger Embedded Studio for ARM[6]

Segger Embedded Studio is a paid tool, but freely available for development kits evaluation (No commercial-use), such as the DWM3001-CDK or nRF52xxx-DKs.

**Warning:** Segger Embedded Studio version 6.x presented issues when building projects using "__vprintf.h", so the recommended version is 5.x. The project was tested with Segger Embedded Studio version 5.60a.

### 6.1.2 Hardware

One of the Qorvo development Kits, ready for Nearby Interaction:

- DWM3000EVB (connected to a Nordic nRF52xxx-DK)
- DWM3001CDK

More info at: UWB SOLUTIONS COMPATIBLE WITH APPLE'S U1 CHIP[7]

[6] https://www.segger.com/downloads/embedded-studio/
[7] https://www.qorvo.com/innovation/ultra-wideband/products/uwb-solutions-compatible-with-apple-u1

## 6.2 Building

Open Segger Embedded Studio, click on File→Open Solution (Ctrl+Shift+O), navigate to the “QNI-2.0.5” folder and open the FreeRTOS folder, it has a list of sub-folders, one for each supported device.

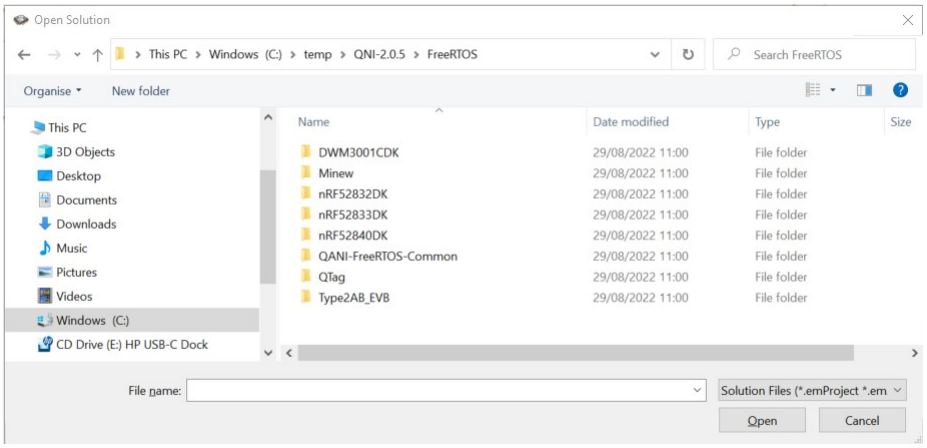


Fig. 6.1: Choose the correct SES solution for the Qorvo device

Open the folder that matches the development kit and open the “/ses/” folder inside, it contains the project file “.emProject”. Select it and click on “Open”

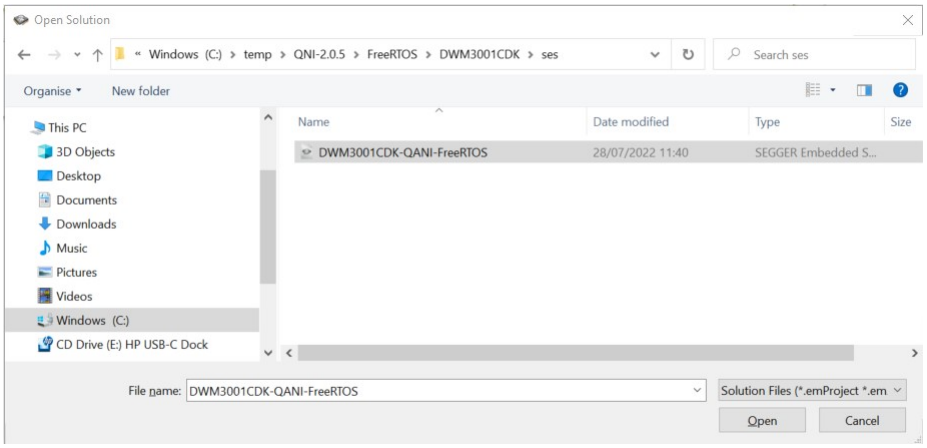


Fig. 6.2: Select the project file and click on “Open”

In the Solution structure on the left, right click on the project name and select “Build” (or select the project and press “F7”).

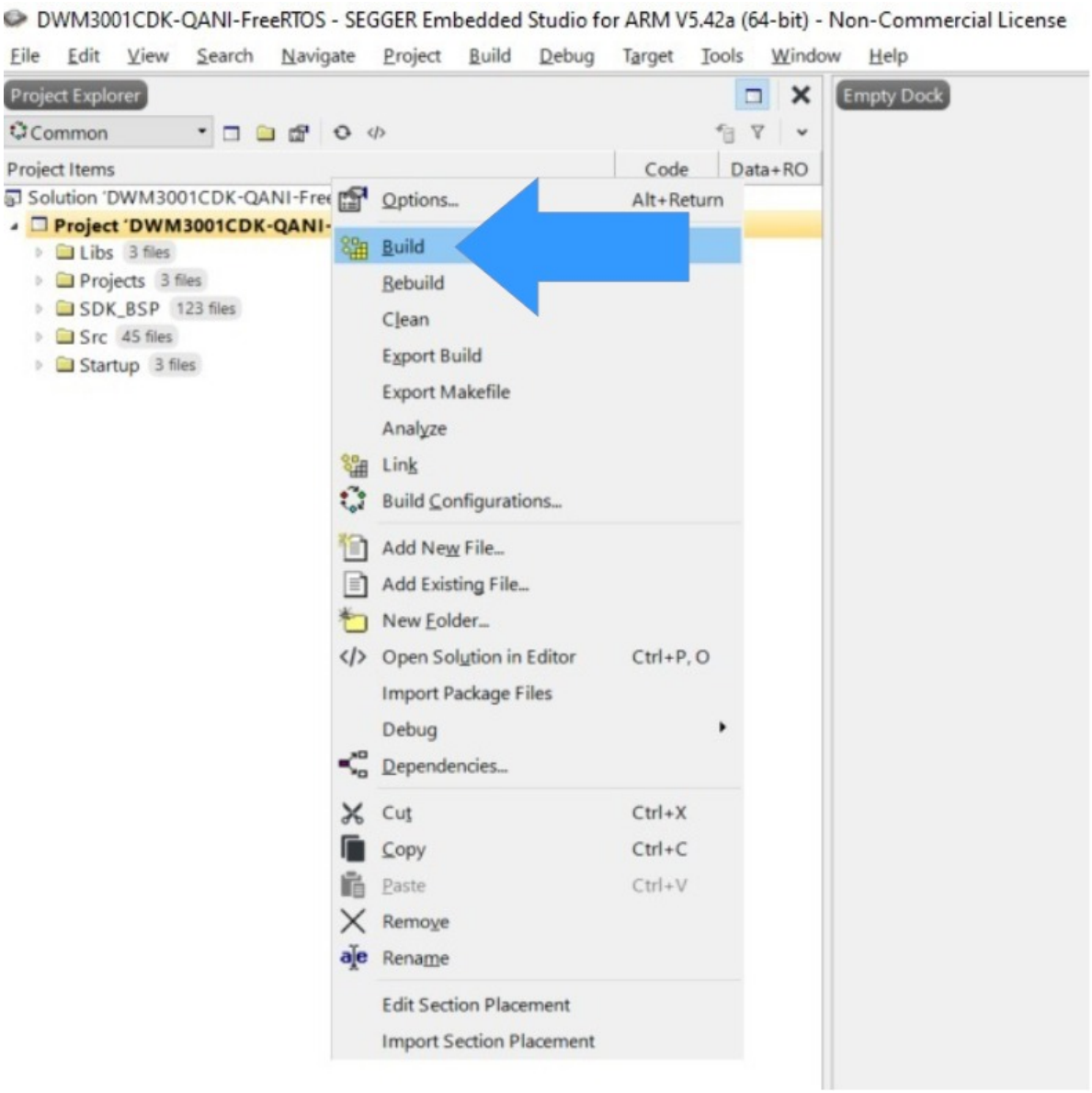


Fig. 6.3: Right click on the project name and select “Build”

When finished, the Output window (bottom in the middle) will show the build information (*Build information when successfully built*).

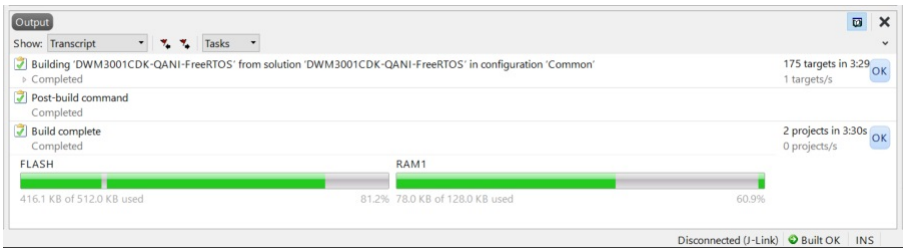


Fig. 6.4: Build information when successfully built

## 6.3 Debugging

With the project building debug can be started. Connect the development kit into a free USB port of the PC running Segger Embedded Studio, it must be connected via the J-Link USB. Then click on Debug→Go, the project will be flashed to the development kit and Segger will switch to Debug Mode (*Segger Debug Mode. Disassembly on the left, running code on the right.*).

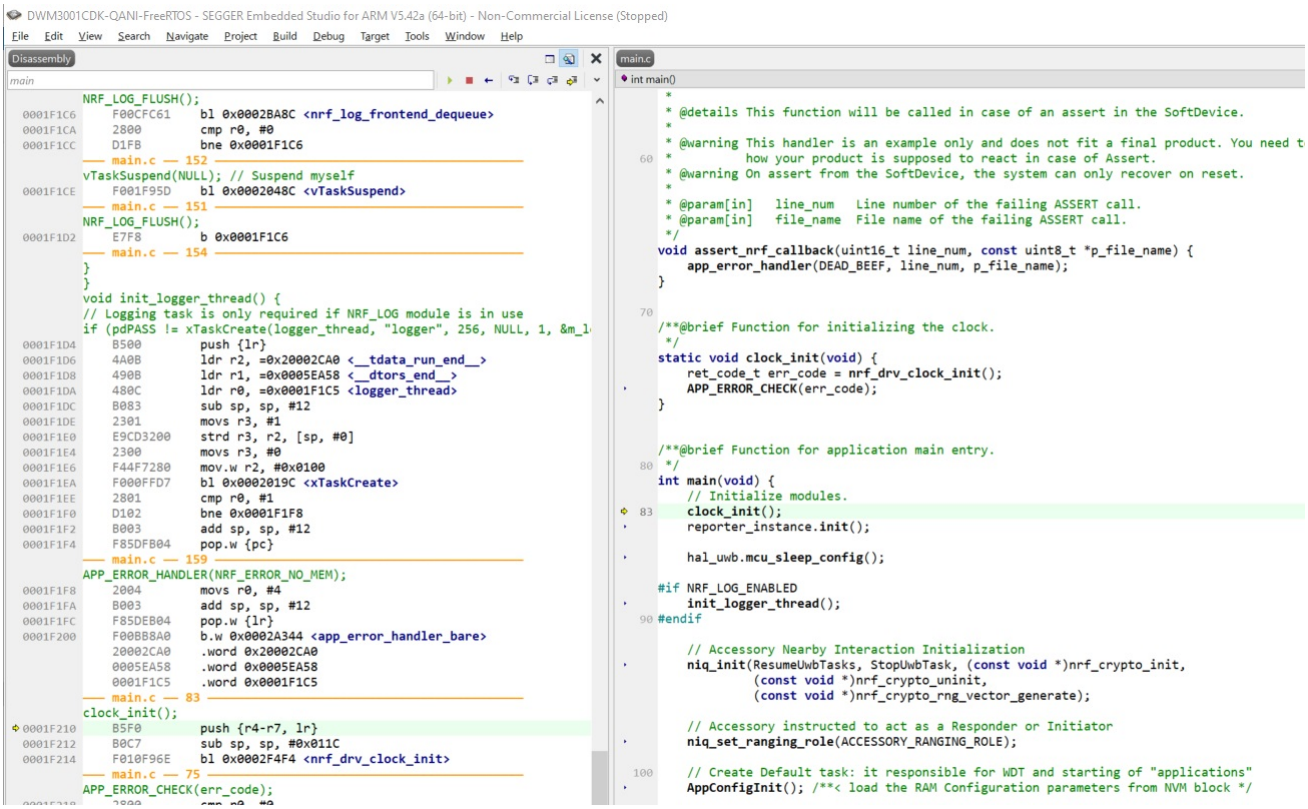


Fig. 6.5: Segger Debug Mode. Disassembly on the left, running code on the right.

The window in the middle will show the code in execution, on the left the functions broken in assembly lines. The execution can be controlled by the controls shown in (*Debug controls, Run, Stop, Reset and execution in steps.*)

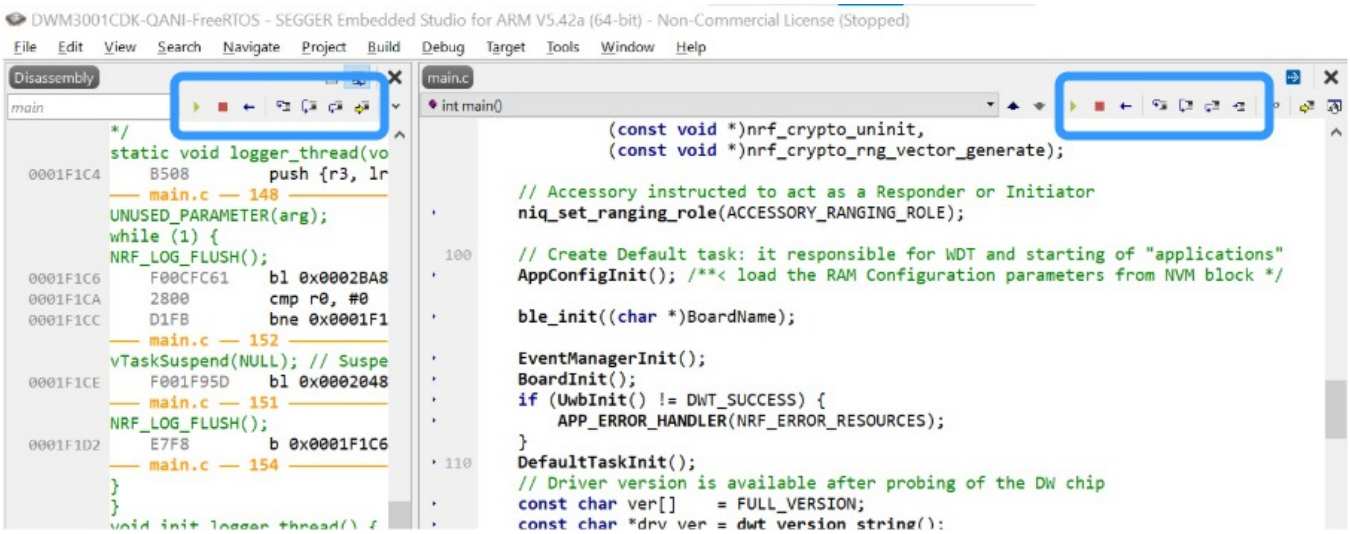


Fig. 6.6: Debug controls, Run, Stop, Reset and execution in steps.

Breakpoints can be added to the code, and when the execution reachs the line with a breakpoint it will be paused.

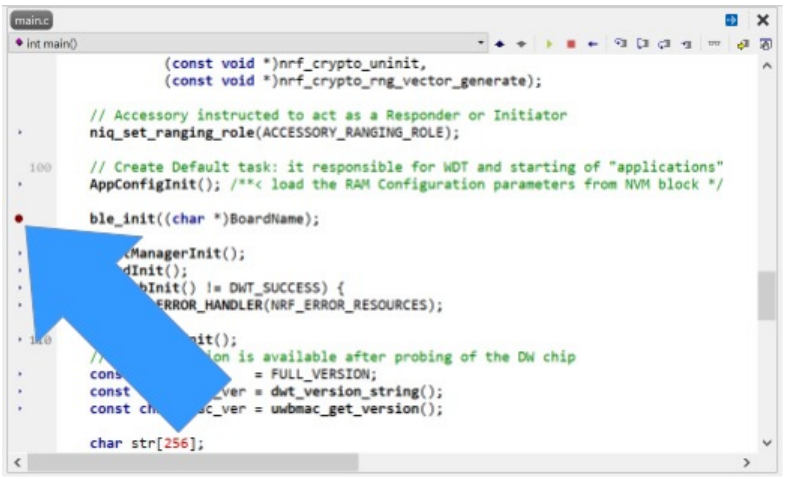


Fig. 6.7: Click on the left of a code line to set a breakpoint (or press F9).

To view all breakpoints click on Debug→Breakpoints→Breakpoints (Ctrl+Alt+B), a window on the right will list all breakpoint.

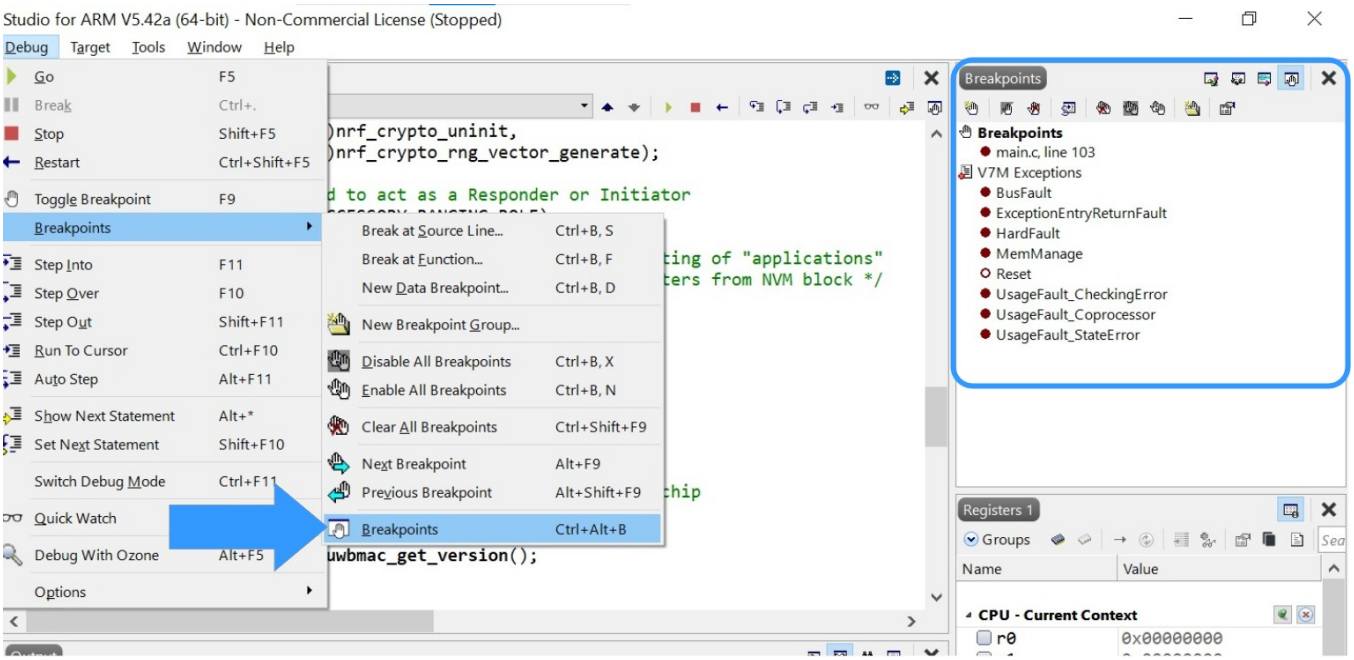


Fig. 6.8: View breakpoints and the breakpoint window on the left.

The QNI Projects use RTT to output important info and debug lines, when in debugging mode you can see the RTT output in the Debug Terminal (window on the bottom middle)

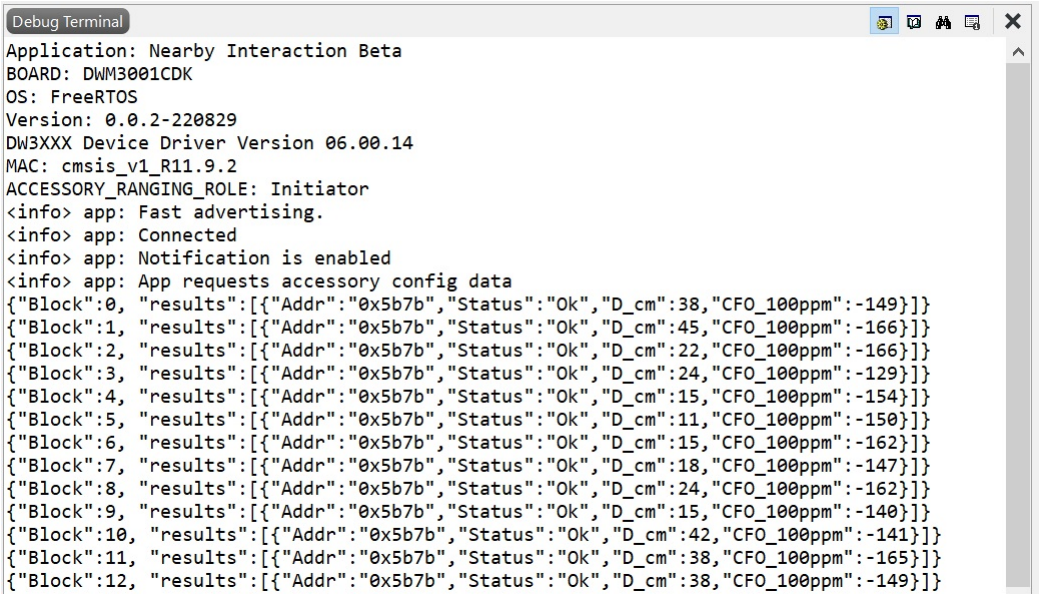


Fig. 6.9: Debug Terminal showing RTT output.

# 7 Referenced Documents


1. Qorvo Nearby Interaction resources
   https://www.qorvo.com/feature/uwb-solutions-compatible-with-apple-u1
2. Apple® Nearby Interaction Source code example
   https://developer.apple.com/documentation/nearbyinteraction/implementing_spatial_interactions_with_third-party_accessories
3. Apple® Nearby Interaction Accessory Protocol Specification Release R2 (login required)
   https://developer.apple.com/nearby-interaction/specification/
4. DW3000 and QM33100 Qorvo Nearby Interaction Accessory Developer Guide
   https://www.qorvo.com/products/d/da008212
5. Nordic Semiconductors developer's site
   https://infocenter.nordicsemi.com

# 8 Revision History

| Version | Date | Comment |
|---|---|---|
| A | 2022-09-01 | Initial version |
| B | 2022-12-21 | iOS 16 Camera Assistance update |
| C | 2023-03-29 | Expanding Project Information (BLE, NIQ, AR, GUI)<br>Different behaviors for iPhone14 |
| D | 2024-01-22 | References to iPhone 15 |
| E | 2025-01-15 | App version update |

# 9 Contact Information

For the latest specifications, additional product information, worldwide sales and distribution locations:
Web: www.qorvo.com
Tel: 1-844-890-8163
Email: customer.support@qorvo.com

# 10 Important Notice

The information contained in this Data Sheet and any associated documents (“Data Sheet Information”) is believed to be reliable; however, Qorvo makes no warranties regarding the Data Sheet Information and assumes no responsibility or liability whatsoever for the use of said information. All Data Sheet Information is subject to change without notice. Customers should obtain and verify the latest relevant Data Sheet Information before placing orders for Qorvo® products. Data Sheet Information or the use thereof does not grant, explicitly, implicitly or otherwise any rights or licenses to any third party with respect to patents or any other intellectual property whether with regard to such Data Sheet Information itself or anything described by such information.

DATA SHEET INFORMATION DOES NOT CONSTITUTE A WARRANTY WITH RESPECT TO THE PRODUCTS DESCRIBED HEREIN, AND QORVO HEREBY DISCLAIMS ANY AND ALL WARRANTIES WITH RESPECT TO SUCH PRODUCTS WHETHER EXPRESS OR IMPLIED BY LAW, COURSE OF DEALING, COURSE OF PERFORMANCE, USAGE OF TRADE OR OTHERWISE, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Without limiting the generality of the foregoing, Qorvo® products are not warranted or authorized for use as critical components in medical, life-saving, or life-sustaining applications, or other applications where a failure would reasonably be expected to cause severe personal injury or death. Applications described in the Data Sheet Information are for illustrative purposes only. Customers are responsible for validating that a particular product described in the Data Sheet Information is suitable for use in a particular application.

FiRa, FiRa Consortium, the FiRa logo, the FiRa Certified logo, and FiRa tagline are trademarks or registered trademarks of FiRa Consortium or its licensor(s)/ supplier(s) in the US and other countries and may not be used without permission. All other trademarks, service marks, and product or service names are trademarks or registered trademarks of their respective owners.