RFID SDK for MAUI Developer Guide

Contents

Overview 2

Environment Setup 3

Create MAUI iOS Project 3

Namespace 8

API Calls 8

- Query SDK Version 8
- Set Operation Mode 8
- Get Available Reader List 9
- Connect/Disconnect RFID Reader 9
- Start/Stop Inventory 9
- Start/Stop Tag Locating 10
- Start/Stop Trigger Configuration 11
- Set Batch Mode Configuration 13
- Set Unique Tag Report 13
- Set Tag Report Configuration 14
- Set Regulatory Configuration 15
- Set Antenna Configuration 16
- Set Singulation Configuration 17
- Set Device Mode 18
- Access Operation Read Tags 18
- Access Operation Write Tags 19
- Access Operation Lock Tags 19
- Access Operation Kill Tags 20

API Events 21

- Activity Events 21
  - Appeared 21
  - Disappeared 22
  - Connected 23
  - Disconnected 24
  - TagDataEvent 25

ProximityPercent 26
OperationBatchMode 27
TriggerNotifyEvent 28
Action Status Events 29
OperationEndSummary 29
Temperature 30
Power 31
Database 31
Radio 32
OperationStart 32
OperationStop 33
WLAN 34
WLAN Scan Event 34
WLAN Scan List 35
WLAN Enable/ Disable 36
Get WLAN Status 37
Get WLAN Profile List 38
Add WLAN Profile 39
Save WLAN Profile 40
Remove WLAN Profile 41
Connect WLAN Profile 42
Get WLAN Certificates List 43
Disconnect WLAN Profile 44
Known Issues 45
Appendix 45

Overview

This document provides step-by-step instructions on developing MAUI Framework based RFID applications for iOS with Visual Studio 2019.

Environment Setup

Please refer the instructions provided below for configuring development environment in the respective platform.

Windows:

Install Visual Studio on Windows computer

https://learn.microsoft.com/en-us/visualstudio/install/install-visual-studio?view=vs-2022

Follow instruction in the provided link below to configure MAUI.

https://learn.microsoft.com/en-us/dotnet/maui/deployment/?view=net-maui-8.0#ios

Additionally, follow instructions below in linking to a MAC which is mandatory requirement.

https://learn.microsoft.com/en-us/dotnet/maui/ios/pair-to-mac?view=net-maui-8.0

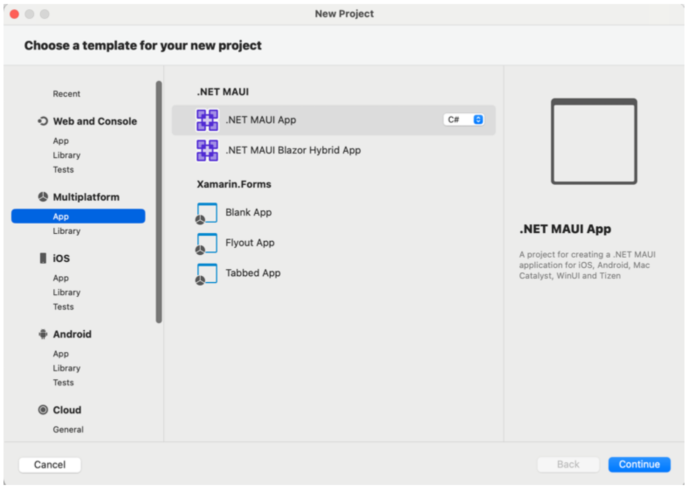
Mac OS:

Follow instructions in the provided link to install Visual Studio on Mac OS.

https://learn.microsoft.com/en-us/visualstudio/install/install-visual-studio?view=vs-2022

Create MAUI iOS Project

1. Open Visual Studio 2022 IDE, create the .NET MAUI App application by following the wizard. And select .NET 8.0 as a Target Framework.

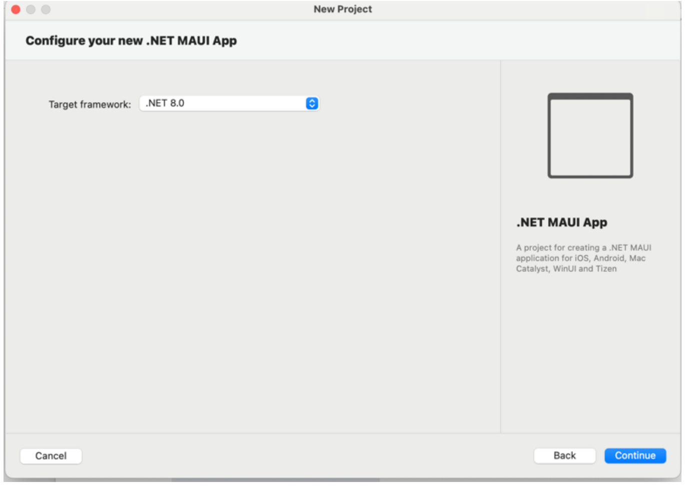


New Project

Choose a template for your new project

.NET MAUI App

A project for creating a .NET MAUI application for iOS, Android, Mac Catalyst, WinUI and Tizen



New Project

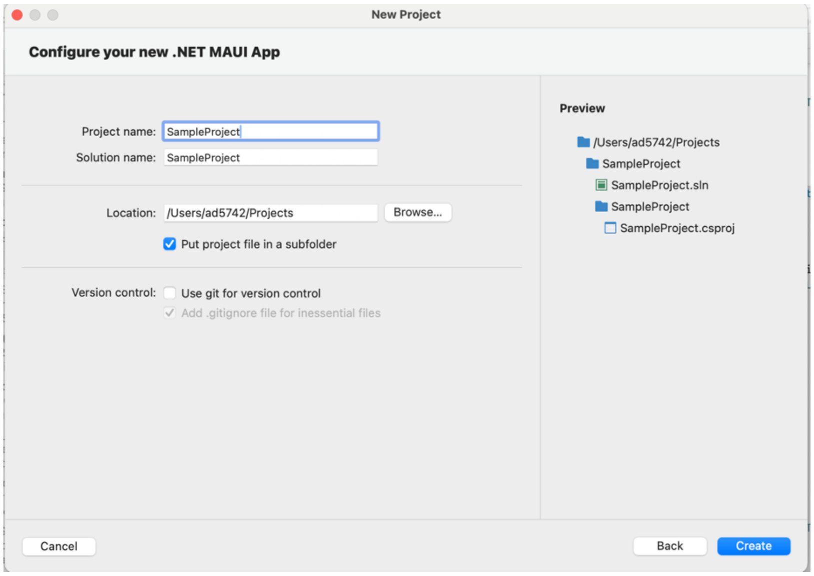
Configure your new .NET MAUI App

Target framework: .NET 8.0

.NET MAUI App

A project for creating a .NET MAUI application for iOS, Android, Mac Catalyst, WinUI and Tizen

2. Provide a *Project Name* and create the project



New Project

Configure your new .NET MAUI App

Project name: SampleProject

Solution name: SampleProject

Location: /Users/ad5742/Projects

☑ Put project file in a subfolder

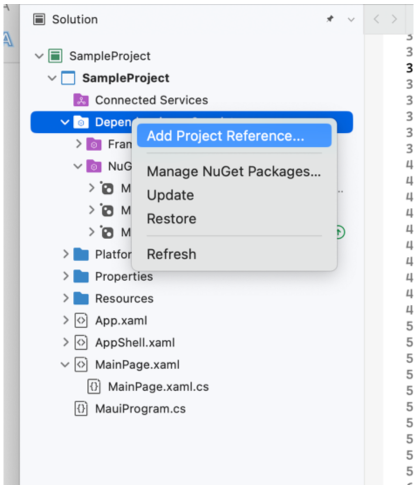
Version control: □ Use git for version control

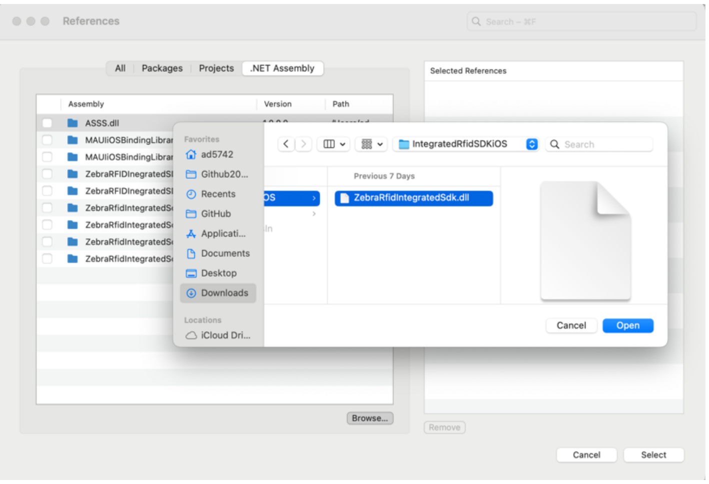
✓ Add .gitignore file for inessential files

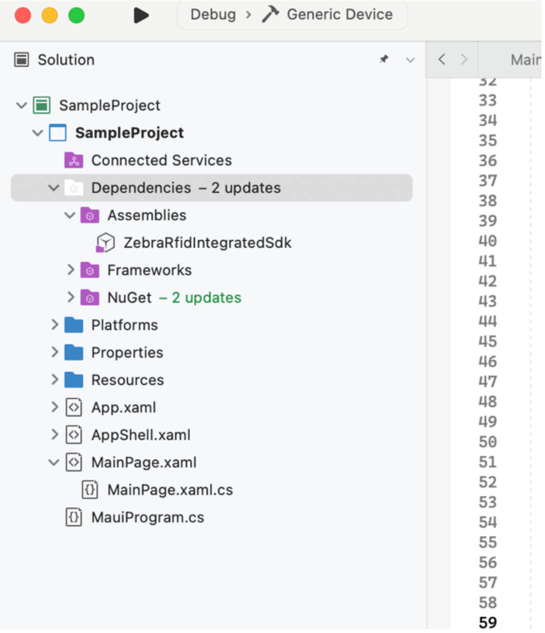
Preview

/Users/ad5742/Projects
SampleProject
SampleProject.sln
SampleProject
SampleProject.csproj

3. Add references as .Net Assembly
    - Right click on *Dependencies* folder and select *Add Project Reference(ZebraRfidIntegratedSdk.dll)*



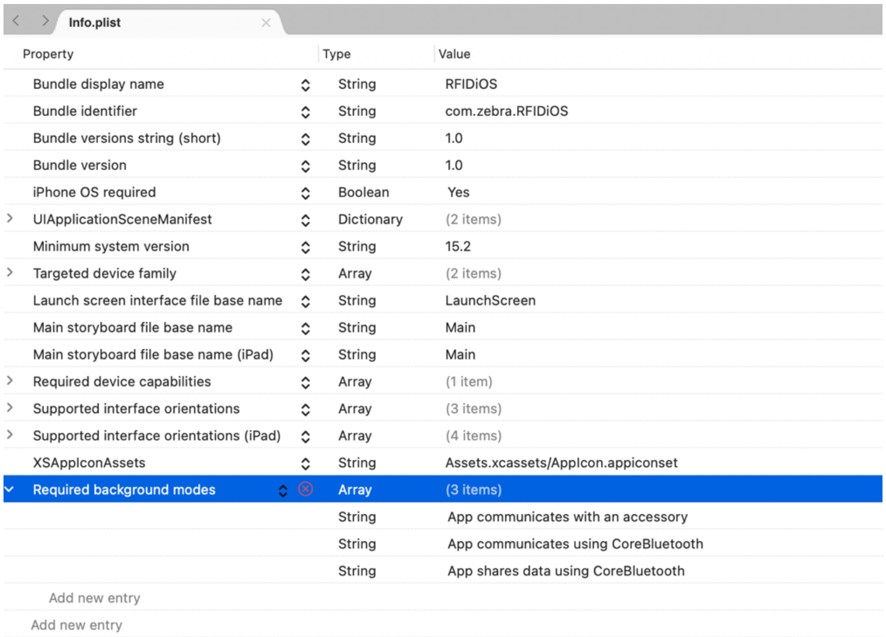



Debug › Generic Device
Solution
Main
SampleProject
SampleProject
Connected Services
Dependencies – 2 updates
Assemblies
ZebraRfidIntegratedSdk
Frameworks
NuGet – 2 updates
Platforms
Properties
Resources
App.xaml
AppShell.xaml
MainPage.xaml
MainPage.xaml.cs
MauiProgram.cs

4. Update the Info.plist as described here

Add following entries under *Required background modes* property

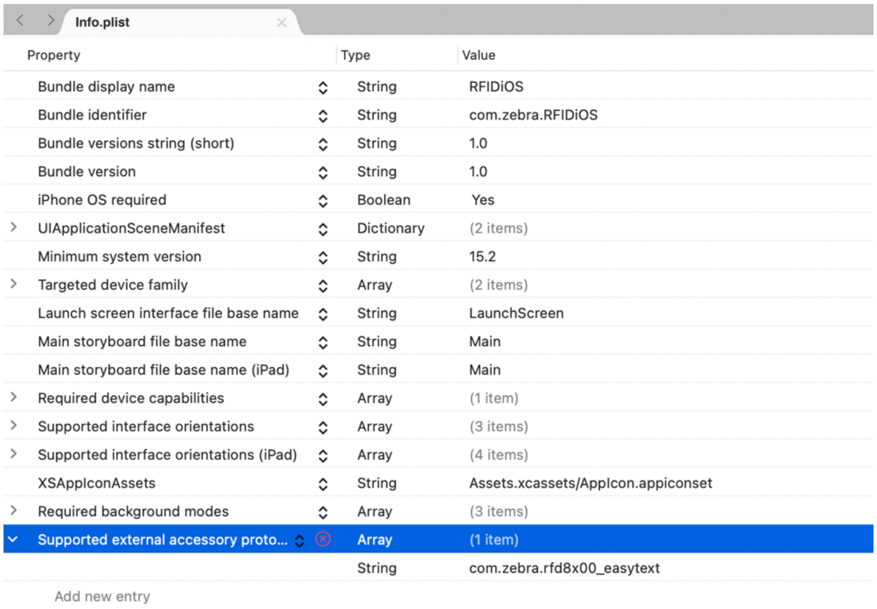
- *App communicates with an accessory*
- *App communicates using CoreBluetooth*
- *App shares data using CoreBluetooth*



Property	Type	Value
Bundle display name	String	RFIDiOS
Bundle identifier	String	com.zebra.RFIDiOS
Bundle versions string (short)	String	1.0
Bundle version	String	1.0
iPhone OS required	Boolean	Yes
UIApplicationSceneManifest	Dictionary	(2 items)
Minimum system version	String	15.2
Targeted device family	Array	(2 items)
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
Main storyboard file base name (iPad)	String	Main
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)
Supported interface orientations (iPad)	Array	(4 items)
XSAppIconAssets	String	Assets.xcassets/AppIcon.appiconset
Required background modes	Array	(3 items)
	String	App communicates with an accessory
	String	App communicates using CoreBluetooth
	String	App shares data using CoreBluetooth
Add new entry		
Add new entry		

Add following string under *Supported external accessory protocols* property.

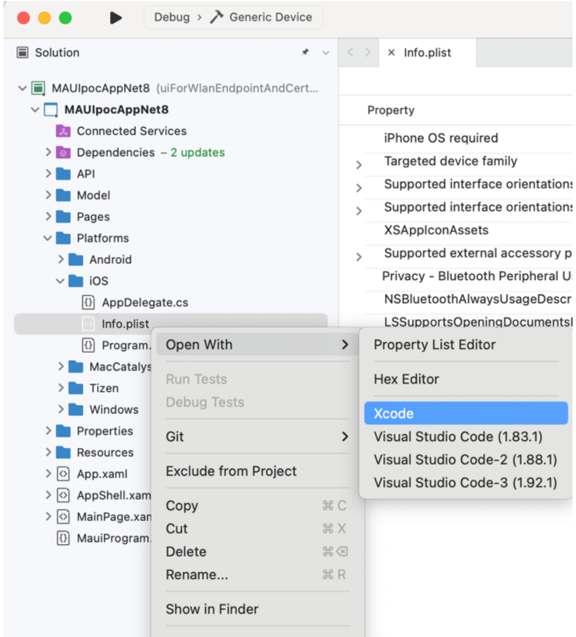
- “com.zebra.rfd8x00_easytext”



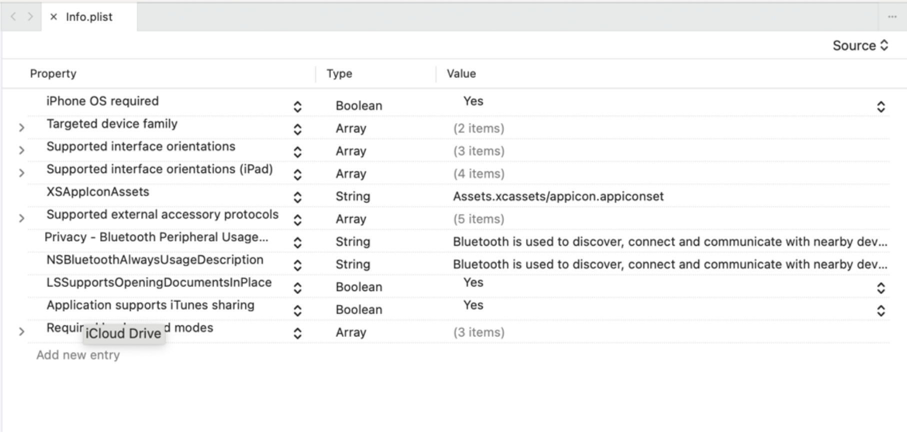
Property	Type	Value
Bundle display name	String	RFIDiOS
Bundle identifier	String	com.zebra.RFIDiOS
Bundle versions string (short)	String	1.0
Bundle version	String	1.0
iPhone OS required	Boolean	Yes
UIApplicationSceneManifest	Dictionary	(2 items)
Minimum system version	String	15.2
Targeted device family	Array	(2 items)
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
Main storyboard file base name (iPad)	String	Main
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)
Supported interface orientations (iPad)	Array	(4 items)
XSAppIconAssets	String	Assets.xcassets/AppIcon.appiconset
Required background modes	Array	(3 items)
Supported external accessory proto...	Array	(1 item)
	String	com.zebra.rfd8x00_easytext
Add new entry		

Add *NSBluetoothAlwaysUsageDescription* property

- To add this property, open *Info.plist* file through XCode
- Right click on *Info.plist* file and select *Open With → Xcode*



- Select *Privacy – Bluetooth Always Usage Description* property and add "Bluetooth is used to discover, connect and communicate with nearby devices" string as the property value



Property	Type	Value
iPhone OS required	Boolean	Yes
Targeted device family	Array	(2 items)
Supported interface orientations	Array	(3 items)
Supported interface orientations (iPad)	Array	(4 items)
XSAppIconAssets	String	Assets.xcassets/appicon.appiconset
Supported external accessory protocols	Array	(5 items)
Privacy - Bluetooth Peripheral Usage…	String	Bluetooth is used to discover, connect and communicate with nearby dev…
NSBluetoothAlwaysUsageDescription	String	Bluetooth is used to discover, connect and communicate with nearby dev…
LSSupportsOpeningDocumentsInPlace	Boolean	Yes
Application supports iTunes sharing	Boolean	Yes
Requir… d modes	Array	(3 items)

- Save and close the opened Xcode file

Namespace

Import the RFID SDK namespace before making API calls.

```
using ZebraRfidSdk;
```

API Calls

Query SDK Version

Version information could be queried as follows.

```
//Create an instance of the RfidSDK
RfidSdk rfidSdk = new RfidSdk();

//Get the SDK version
string version = rfidSdk.Version;
```

Set Operation Mode

Set operation mode of the reader.

```
//Create an instance of the Readers
Readers readerManager = rfidSdk.ReaderManager;

//Set Operation Mode. Communicate with RFID readers in MFi mode
readerManager.SetOperationMode(OpMode.OPMODE_MFI);
```

Available Operation Modes

- OPMODE_MFI
- OPMODE_BTLE
- OPMODE_ALL

Get Available Reader List.

Query paired device list as follows. Reader must be paired with the iOS device via Bluetooth before query action.

```
//Get available readers list
//readerManager is a Readers object that can be obtained via an instance of the
RfidSDK
List<Reader> readerList = readerManager.GetReaders();
```

Connect/Disconnect RFID Reader

Connect to an available reader.

```
//Connect to an available reader. As an example, connect to the first available
reader. The event Connected will be triggered after the reader is connected.
readerList[0].Connect();
```

Disconnected from the connected reader.

```
//Disconnect from a given reader. The event Disconnected will be triggered after
the reader is disconnected.
readerList[0].Disconnect();
```

Start/Stop Inventory

RFID tag reading can be started as follows. Once started, tags in the range will be read continuously.

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

//Start reading available RFID tags. The event TagDataEvent will be triggered after
the Inventory starts.
connectedReader.Actions.Inventory.Start();
```

RFID tag reading cycle can be terminated as follows.

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

//Stop reading RFID tags
connectedReader.Actions.Inventory.Stop();
```

Start/Stop Tag Locating

Tag locating can be started as follows.

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

//Locate tags with the connectedReader. The event ProximityPercent will be
triggered after the TagLocate starts.

connectedReader.Actions.TagLocate.Start(tag_epc_id);
```

tag_epc_id – string – id of a tag to be located

Stop locating tags.

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

//Stop locating tags
connectedReader.Actions.TagLocate.Stop();
```

Start/Stop Trigger Configuration

Set *Start Trigger Configuration* to the reader.

```
//Set Trigger Configurations
StartTriggerConfiguration configuration = new StartTriggerConfiguration();
configuration.RepeatMonitoring = true;
configuration.StartDelay = 1;
configuration.StartOnHandheldTrigger = true;
configuration.TriggerType = TriggerType.TRIGGERTYPE_PRESS

///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

//Set start trigger configurations
connectedReader.Configuration.StartTriggerConfiguration = configuration;
```

Available options for Start Trigger configuration

- RepeatMonitoring : bool – Repeat monitoring for start trigger after stop of operation
- StartOnHandheldTrigger : bool – Start of an operation based on a physical trigger
- StartDelay : int – Delay (in miliseconds) of start of operation
- TriggerType : TriggerType – Trigger type of a physical trigger
    - TRIGGERTYPE_RELEASE
    - TRIGGERTYPE_PRESS

Set *Stop Trigger Configuration* to the reader.

```
//Set configurations
StopTriggerConfiguration configuration = new StopTriggerConfiguration();
configuration.StopOnAccessCount = true;
configuration.StopInventoryCount = 1;
configuration.StopAccessCount = 1;

///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

//Set stop trigger configurations
connectedReader.Configuration.StopTriggerConfiguration = configuration;
```

Available options for Stop Trigger Configuration

- StopOnAccessCount : bool
- StopOnHandheldTrigger : bool
- StopOnInventoryCount : bool
- StopOnTagCount : bool
- StopOnTimeOut : bool
- DurationMilliSecond : int -
- StopAccessCount : int - Stop of an operation based on a specified number of access rounds completed
- StopInventoryCount : int - Stop of an operation based on a specified number of inventory rounds completed
- StopTagCount : int - Stop of an operation based on a specified number of tags inventoried
- StopTimeOut : int – Stop of an operation based on a specified timeout (in milliseconds)
- TriggerType : TriggerType - Trigger type of a physical trigger
    - TRIGGERTYPE_RELEASE
    - TRIGGERTYPE_PRESS

Set Batch Mode Configuration

Batch Mode configurations can be applied as follows.

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

//Set Batch mode congfiguration
connectedReader.Configuration.BatchModeConfiguration = BatchMode.AUTO;
```

Supported values for Batch Mode

- Auto
- Disable
- Enable

Set Unique Tag Report

Unique Tag Report can be applied as follows.

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

//Set Unique Tag Report
connectedReader.Configuration.UniqueTagReport = true;
```

Supported values are *true* or *false*.

Set Tag Report Configuration

Tag Report Configuration can be applied as follows.

```
//Set Tag Report Configururations
TagReportConfiguration configuration = new TagReportConfiguration();
configuration.ChannelIdx = true;
configuration.FirstSeenTime = true;
configuration.LastSeenTime = true;
configuration.Pc = true;
configuration.Rssi = true;
configuration.Phase = true;
configuration.TagSeenCount = true;

///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

connectedReader.Configuration.TagReportConfiguration = configuration;
```

Available options for *Tag Report Configuration*

- `ChannelIdx`: bool
- FirstSeenTime : bool
- LastSeenTime : bool
- Pc : bool
- Phase : bool
- Rssi : bool
- TagSeenCount : bool

Set Regulatory Configuration

Regulatory Configuration can be applied as follows.

```
RegulatoryConfig regulatoryConfig = new RegulatoryConfig();

// get list of supported regions for a connected reader
List<RegionInformation> supportedRegions = connectedReader.SupportedRegions;

foreach (RegionInformation supportRegion in supportedRegions)
{
    // check whether USA region is supported
    if (supportRegion.RegionCode == "USA")
    {
         // set configuration to USA region
        regulatoryConfig.RegionCode = "USA";
        //EnableChannels are hardcoded because of an issue in getting supported
Channels for a specific RegionInformation
        regulatoryConfig.EnableChannels = new object[] { "915750", "915250",
"903250" };
        regulatoryConfig.HoppingConfig = HoppingConfig.HOPPINGCONFIG_DEFAULT;
        regulatoryConfig.HoppingOn = true;
        break;
    }

}

///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

//Set Regulatory Configurations
connectedReader.Configuration.RegulatoryConfig = regulatoryConfig;
```

Available options for *Regulatory Configuration*

- HoppingOn : bool
- EnableChannels : object[] – Set of enabled channels
- HoppingConfig : HoppingConfig
    - HOPPINGCONFIG_DEFAULT
    - HOPPINGCONFIG_ENABLED
    - HOPPINGCONFIG_DISABLED
- RegionCode : string - Code of selected region

Set Antenna Configuration

Antenna Configuration can be applied to a connected reader as follows

```
RfidSdk sdkInstance = new RfidSdk();

AntennaConfiguration antennnaConfig = new AntennaConfiguration();
antennnaConfig.AntennaPower = 300;
//Supported Link Profiles can be seen in Appendix
antennnaConfig.LinkProfile = 1;
antennnaConfig.Tari = 6250;
antennnaConfig.DoSelect = true;

///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

//Set Antenna configuration
connectedReader.Configuration.Antennas.AntennaConfiguration = antennnaConfig;
```

Available options for *Antenna Configuration*

- AntennaPower : int - Output power level(in 0.1 dbm units)
- LinkProfile : int - Index of selected link profile
- Tari : int - Type-A reference interval
- DoSelect : bool - Specifies whether Antenna pre-filters can be applied or not.

Set Singulation Configuration

Singulation Control configurations can be applied as follows.

```
//Set Singulation Configurations
SingulationControl singulationControl = new SingulationControl();
singulationControl.SelectedFlag = SLFlag.SLFLAG_ALL;
singulationControl.Session = Session.SESSION_S0;
singulationControl.State = InventoryState.INVENTORYSTATE_A;
singulationControl.TagPopulation = 200;

///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

connectedReader.Configuration.Antennas.SingulationControl = singulationControl;
```

Available options for Singulation Configuration.

- SelectedFlag : SLFlag – Selected flag
  - SLFLAG_ASSERTED,
  - SLFLAG_DEASSERTED,
  - SLFLAG_ALL
- Session : Session – Session number to use for inventory operation
  - SESSION_S0,
  - SESSION_S1,
  - SESSION_S2,
  - SESSION_S3
- State : InventoryState – Target inventory state
  - INVENTORYSTATE_A,
  - INVENTORYSTATE_B,
  - INVENTORYSTATE_AB_FLIP
- TagPopulation : int – an estimate of the tag population in view of the RF field of the antenna

Set Device Mode

Set device mode of the reader.

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

//Set Device Mode
connectedReader.Configuration.SetDeviceMode(DeviceMode.RFID);
```

Available Device Modes;

- RFID
- BARCODE

Access Operation Read Tags

Following values should be passed as arguments to *AccessOperationsReadTag* API and it will return a TagData object.

tagId - string
tagAccessPassword - string
byteCount - short
offset - short
memoryBank – MemoryBank

- MEMORYBANK_EPC
- MEMORYBANK_TID
- MEMORYBANK_USER
- MEMORYBANK_RESV
- MEMORYBANK_NONE
- MEMORYBANK_ACCESS
- MEMORYBANK_KILL

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

TagData tagdataObject = connectedReader.AccessOperationsReadTag(tagId,
tagAccessPassword, byteCount, offset, memoryBank);
```

Access Operation Write Tags

Following values should be passed as arguments to *AccessOperationsWriteTag* API and it will return a boolean value whether the write operation is successful or not.

tagId - string
tagAccessPassword - string
tagData - string
offset - short
memoryBank - MemoryBank

- MEMORYBANK_EPC
- MEMORYBANK_TID
- MEMORYBANK_USER
- MEMORYBANK_RESV
- MEMORYBANK_NONE
- MEMORYBANK_ACCESS
- MEMORYBANK_KILL

blockWrite - bool

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

bool tagWriteResult = connectedReader.AccessOperationsWriteTag(tagId,
tagAccessPassword, tagData, offset, memoryBank, blockWrite);
```

Access Operation Lock Tags

Following values should be passed as arguments to *AccessOperationsLockTag* API and it will return a boolean value whether the lock operation is successful or not.

tagId - string
tagAccessPassword - string
memoryBank - MemoryBank

- MEMORYBANK_EPC
- MEMORYBANK_TID
- MEMORYBANK_USER
- MEMORYBANK_RESV

- MEMORYBANK_NONE
- MEMORYBANK_ACCESS
- MEMORYBANK_KILL

lockPrivilege

- READ_WRITE
- PERMANENT_LOCK
- PERMANENT_UNLOCK
- UNLOCK

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

bool tagLockResult = connectedReader.AccessOperationsLockTag(tagId,
tagAccessPassword, memoryBank, lockPrivilege);
```

Access Operation Kill Tags

Following values should be passed as arguments to *AccessOperationsKillTag* API and it will return a boolean value whether the kill operation is successful or not.

tagId - string
tagAccessPassword - string

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

bool tagKillResult = connectedReader.AccessOperationsKillTag(tagId,
tagAccessPassword);
```

API Events

Activity Events

Appeared

This event is triggered when a reader appeared.

```
//readerManager is a Readers object that can be obtained via an instance of the
RfidSDK

//Subscribes for the Appeared event
readerManager.Appeared += ReaderManager_Appeared;

// Event handler of Reader appeared event
void ReaderManager_Appeared(Reader readerInfo)
{
    try
    {
        Console.WriteLine("Reader Appeared reader id" + readerInfo.Id);
        Console.WriteLine("Reader Appeared reader name" + readerInfo.Name);
        Console.WriteLine("Reader Appeared reader model" + readerInfo.Model);
        Console.WriteLine("Reader Appeared reader status" + readerInfo.IsActive);

    }
    catch (Exception e)
    {
        Console.WriteLine("Exception " + e.Message);
    }

}
```

readerInfo – a *Reader* object containing information of the appeared reader.

Disappeared

This event triggers when a reader disappeared.

```
//readerManager is a Readers object that can be obtained via an instance of the
RfidSDK

//Subscribes for the Disappeared event

readerManager.Disappeared += ReaderManager_Disappeared;

// Event handler of Reader disappeared event
 void ReaderManager_Disappeared(int readerID)
{
    try
    {
        Console.WriteLine("Reader Disappeared" + readerID);

    }
    catch (Exception e)
    {
        Console.WriteLine("Exception " + e.Message);
    }

}
```

readerID - ID of the disappeared reader.

Connected

This event triggers when an available reader is connected

```
//readerManager is a Readers object that can be obtained via an instance of the
RfidSDK

//Subscribes for the Connected event

readerManager.Connected += ReaderManager_Connected;

/// Event handler of Reader connected event
void ReaderManager_Connected(Reader reader)
{
    try
    {
        Console.WriteLine("Reader Connected, reader id: “ + reader.Id);
        Console.WriteLine(“Reader Connected, reader name: " + reader.Name);

    }
    catch (Exception e)
    {
        Console.WriteLine("Exception " + e.Message);
    }

}
```

reader – a Reader object that provides information of the reader connected.

Disconnected

This event triggers when a connected reader is disconnected

```
//readerManager is a Readers object that can be obtained via an instance of the
RfidSDK

//Subscribes for the Disconnected event

readerManager.Disconnected += ReaderManager_Disconnected;

// Event handler of Reader Disconnected event
void ReaderManager_Disconnected(int readerID)
{
    Console.WriteLine("Reader Disconnected, reader id: " + readerID);

}
```

readerID – reader id of the reader disconnected

TagDataEvent

This event triggers when tag data is received

```
//readerManager is a Readers object that can be obtained via an instance of the
RfidSDK

// Subscribes for the event TagDataEvent

readerManager.TagDataEvent += ReaderNotifyDataEvent;

//Event handler of Reader notify tag data event
void ReaderNotifyDataEvent(TagData tagData)
{
    try
    {
        Console.WriteLine("Reader Notify Data Event tag id " + tagData.Id);
        Console.WriteLine("Reader Notify Data Event memory bank" +
tagData.MemoryBank);
        Console.WriteLine("Reader Notify Data Event memory bank data" +
tagData.MemoryBankData);
        Console.WriteLine("Reader Notify Data Event seen count" +
tagData.SeenCount);

    }
    catch (Exception e)
    {
        Console.WriteLine("Exception " + e.Message);
    }

}
```

tagdata – TagData object that provides information of the tag read by the reader.

ProximityPercent

This event will trigger when reception of a proximity notification during on-going tag locating operation from a connected RFID reader.

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///
// Subscribes for the event ProximityPercent
connectedReader.ProximityPercent += Reader_ProximityPercent;

//Event handler of Reader ProximityPercent event
void Reader_ProximityPercent(int proximityPrecentage)
{
    Console.WriteLine("Proximity Percentage " + proximityPrecentage);
}
```

proximityPercentage - provides proximity information as a percentage of the tag from the reader.

OperationBatchMode

This event will trigger when a reader is gone to the batch mode.

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///
// Subscribes for the event OperationBatchmode

readerManager.OperationBatchmode += ReaderManager_OperationBatchmode;

// Event handler of Reader in batchmode.
void ReaderManager_OperationBatchmode(EventStatus eventStatus)
{
    Console.WriteLine("Reader in batch mode");
}
```

Available Event Status

- STATUS_OPERATION_START
- STATUS_OPERATION_STOP
- STATUS_OPERATION_BATCHMODE
- STATUS_OPERATION_END_SUMMARY
- STATUS_TEMPERATURE
- STATUS_POWER
- STATUS_DATABASE
- STATUS_RADIOERROR

TriggerNotifyEvent

This event will trigger when press/release trigger button in the device.

```
///connectedReader is an already connected Reader object that can be obtained via 
the Connected event///
// Subscribes for the event TriggerNotifyEvent

readerManager.TriggerNotifyEvent += ReaderManagerTriggerEvent;

// Event handler of Trigger press/release event.
void ReaderManagerTriggerEvent(int readerID, TriggerType triggerEvent)
{

    Console.WriteLine("Reader id  : " + readerID);
    Console.WriteLine("TriggerType : " + triggerEvent);

}
```

Available Trigger Types

- TRIGGERTYPE_PRESS
- TRIGGERTYPE_RELEASE

Action Status Events

Following events can be registered to get a RFID reader related information.

OperationEndSummary

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

connectedReader.OperationEndSummary += ReaderOperationEndSummaryEvent;

// Event hander for ReaderOperationEndSummaryEvent
void ReaderOperationEndSummaryEvent(OperationEndSummaryEvent endSummary)
{
    Console.WriteLine("Reader Operation End Summary Event, total tags :" +
endSummary.TotalTags);
    Console.WriteLine("Reader Operation End Summary Event, total rounds " +
endSummary.TotalRounds);
    Console.WriteLine("Reader Operation End Summary Event, total time " +
endSummary.TotalTime);

}
```

Temperature

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

connectedReader.Temprature += ReaderTemperatureEvent;

// Event handler for Reader Temperature Event
void ReaderTemperatureEvent(TemperatureEvent temperature)
{
    Console.WriteLine("Reader Temperature Event, event cause :" +
temperature.EventCause);
    Console.WriteLine("Reader Temperature Event, system temperature " +
temperature.SystemTemprature);
    Console.WriteLine("Reader Temperature Event, radio temperature " +
temperature.RadioTemprature);
}
```

Power

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

connectedReader.Power += ReaderPowerEvent;

// Event handler for Reader Power Event
void ReaderPowerEvent(PowerEvent power)
{
    Console.WriteLine("Reader Power Event, power :" + power.Power);
    Console.WriteLine("Reader Power Event, power status :" + power.PowerStatus);
    Console.WriteLine("Reader Power Event, current : " + power.Current);
    Console.WriteLine("Reader Power Event, voltage : " + power.Voltage);

}
```

Database

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

connectedReader.Database += ReaderDatabaseEvent;

// Event handler for Reader Database Event
void ReaderDatabaseEvent(DatabaseEvent database)
{
    Console.WriteLine("Reader Database Event, database status :" +
database.DatabaseStatus);
    Console.WriteLine("Reader Database Event, entries used :" +
database.EntriesUsed);
    Console.WriteLine("Reader Database Event, entries remaining : " +
database.EntriesRemaining);

}
```

Radio

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

connectedReader.Radio += ReaderRadioErrorEvent;

// Event handler for Reader Radio Error Event
void ReaderRadioErrorEvent(RadioErrorEvent radioError)
{
    Console.WriteLine("Reader Radio Error Event, error event status :" +
radioError.EventStatus);
    Console.WriteLine("Reader Radio Error Event, error number :" +
radioError.ErrorNumber);

}
```

OperationStart

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

connectedReader.OperationStart += ReaderOperationStartEvent;

// Event handler for Reader operation start Event
void ReaderOperationStartEvent(EventStatus eventStatus)
{
    //Actions

}
```

OperationStop

```
///connectedReader is an already connected Reader object that can be obtained via
the Connected event///

connectedReader.OperationStop += ReaderOperationStopEvent;

// Event handler for Reader operation stop Event
void ReaderOperationStopEvent(EventStatus eventStatus)
{
    //Actions

}
```

WLAN

WLAN Scan Event

```
// Wifi scan event
        public override void SrfidEventWifiScan(int readerID, srfidWlanScanList wlanScanObject)
      {
         if (wlanScanObject.WlanSsid != null)
         {
            wifiScanListArray.Add(wlanScanObject);
         }
         var handler = WlanScanEvent;
         if (handler != null)
         {
            handler.Invoke(readerID, wlanScanObject);
            wifiScanListArray.Append(wlanScanObject);
         }
     }
```

WLAN Scan List

```
// Get wlan scans list
    public NSMutableArray GetWlanScanList()
    {
        string statusMessage = null;
        NSMutableArray wlanScanList = new NSMutableArray();
        IntPtr availableHandle = wlanScanList.Handle;
        SrfidResult wlanScanListResult = apiInstance.SrfidGetWlanScanList(connectedReaderID, ref
statusMessage);
        wlanScanList = ObjCRuntime.Runtime.GetNSObject<NSMutableArray>(availableHandle);

        if (wlanScanListResult == SrfidResult.Success)
        {
            System.Diagnostics.Debug.WriteLine("Native SrfidGetWlanScanList : Success");
        }
        else if (wlanScanListResult == SrfidResult.ResponseError)
        {
            System.Diagnostics.Debug.WriteLine("SrfidGetWlanScanList ResponseError");
            logsString = "Response Error";
        }
        else if (wlanScanListResult == SrfidResult.InvalidParams)
        {
            System.Diagnostics.Debug.WriteLine("SrfidGetWlanScanList Invalid Prams");
            logsString = "Invalid Parameters";
        }
        else if (wlanScanListResult == SrfidResult.Failure || wlanScanListResult ==
SrfidResult.ResponseTimeout)
        {
            System.Diagnostics.Debug.WriteLine("SrfidGetWlanScanList reder prob");
            logsString = "Reader failure : Response timeout";
        }
        return wlanScanList;
    }
```

WLAN Enable/ Disable

```
// Enable or disable the wifi
        public void RfidWifiEnableDisable(bool state)
        {
            string statusMessage = null;
            SrfidResult wifiState = apiInstance.SrfidWifiEnableDisable(connectedReaderID, state, ref 
statusMessage);

            if (wifiState == SrfidResult.Success)
            {
                System.Diagnostics.Debug.WriteLine("Native SrfidWifiEnableDisable : Success");
                if (state == true)
                {
                    logsString = "WiFi feature enabled Successfully";
                }
                else
                {
                    logsString = "WiFi feature disabled Successfully";
                }
            }
            else if (wifiState == SrfidResult.ResponseError)
            {
                System.Diagnostics.Debug.WriteLine("SrfidWifiEnableDisable ResponseError");
                logsString = "Response Error";
            }
            else if (wifiState == SrfidResult.InvalidParams)
            {
                System.Diagnostics.Debug.WriteLine("SrfidWifiEnableDisable Invalid Prams");
                logsString = "Invalid Parameters";
            }
            else if (wifiState == SrfidResult.Failure || wifiState == SrfidResult.ResponseTimeout)
            {
                System.Diagnostics.Debug.WriteLine("SrfidWifiEnableDisable reder prob");
                logsString = "Reader failure : Response timeout";
            }
        }
```

Get WLAN Status

```
// Get the wifi status (enabled or disabled)
    public string GetWiFiStatus()
    {
      string statusMessage = null;
      srfidGetWifiStatusInfo wlanInfo = new srfidGetWifiStatusInfo();
      IntPtr availableHandle = wlanInfo.Handle;

      SrfidResult wlanScanStatus = SrfidResult.Failure;
      //Retry for 2 times if we get any failure/timeref response
      for (int i = 0; i < 2; i++)
      {
        wlanScanStatus = apiInstance.SrfidGetWifiStatus(connectedReaderID, ref availableHandle, ref statusMessage);
        wlanInfo = ObjCRuntime.Runtime.GetNSObject<srfidGetWifiStatusInfo>(availableHandle);

        if ((wlanScanStatus != SrfidResult.ResponseTimeout) && (wlanScanStatus != SrfidResult.Failure))
        {
          break;
        }
      }

      if (wlanScanStatus == SrfidResult.Success)
      {
        System.Diagnostics.Debug.WriteLine("Native SrfidGetWifiStatus : Success");
        logsString = "Wifi Status: " + wlanInfo.WifiStatus;
        return wlanInfo.WifiStatus;
      }
      else if (wlanScanStatus == SrfidResult.ResponseError)
      {
        System.Diagnostics.Debug.WriteLine("SrfidGetWifiStatus ResponseError");
        logsString = "Response Error";
        return wlanInfo.WifiStatus;
      }
      else if (wlanScanStatus == SrfidResult.InvalidParams)
      {
        System.Diagnostics.Debug.WriteLine("SrfidGetWifiStatus Invalid Prams");
        logsString = "Invalid Parameters";
        return wlanInfo.WifiStatus;
      }
      else if (wlanScanStatus == SrfidResult.Failure || wlanScanStatus == SrfidResult.ResponseTimeout)
      {
        System.Diagnostics.Debug.WriteLine("SrfidGetWifiStatus reder prob");
        logsString = "Reader failure : Response timeout";
        return wlanInfo.WifiStatus;
      }
      else
      {
        return wlanInfo.WifiStatus;
      }
    }
```

Get WLAN Profile List

```
// Get Wlan profiles list
    public SrfidResult GetWlanProfileList(NSMutableArray wlanProfileList)
    {
      string statusMessage = null;
      SrfidResult srfid_result = SrfidResult.Failure;
      for (int i = 0; i < 2; i++)
      {
        srfid_result = apiInstance.SrfidGetWlanProfileList(connectedReaderID, ref wlanProfileList, ref
statusMessage);

        if ((srfid_result != SrfidResult.ResponseTimeout) && (srfid_result != SrfidResult.Failure))
        {
          break;
        }
      }
      if (srfid_result == SrfidResult.Success)
      {
        System.Diagnostics.Debug.WriteLine("SrfidGetWlanProfileList sucess");
      }
      else if (srfid_result == SrfidResult.ResponseError)
      {
        System.Diagnostics.Debug.WriteLine("SrfidGetWlanProfileList SRFID_RESULT_RESPONSE_ERROR");
      }
      else if (srfid_result == SrfidResult.Failure || srfid_result == SrfidResult.ResponseTimeout)
      {
        System.Diagnostics.Debug.WriteLine("SrfidGetWlanProfileList readerProblem");
      }
      return srfid_result;
    }
```

Add WLAN Profile

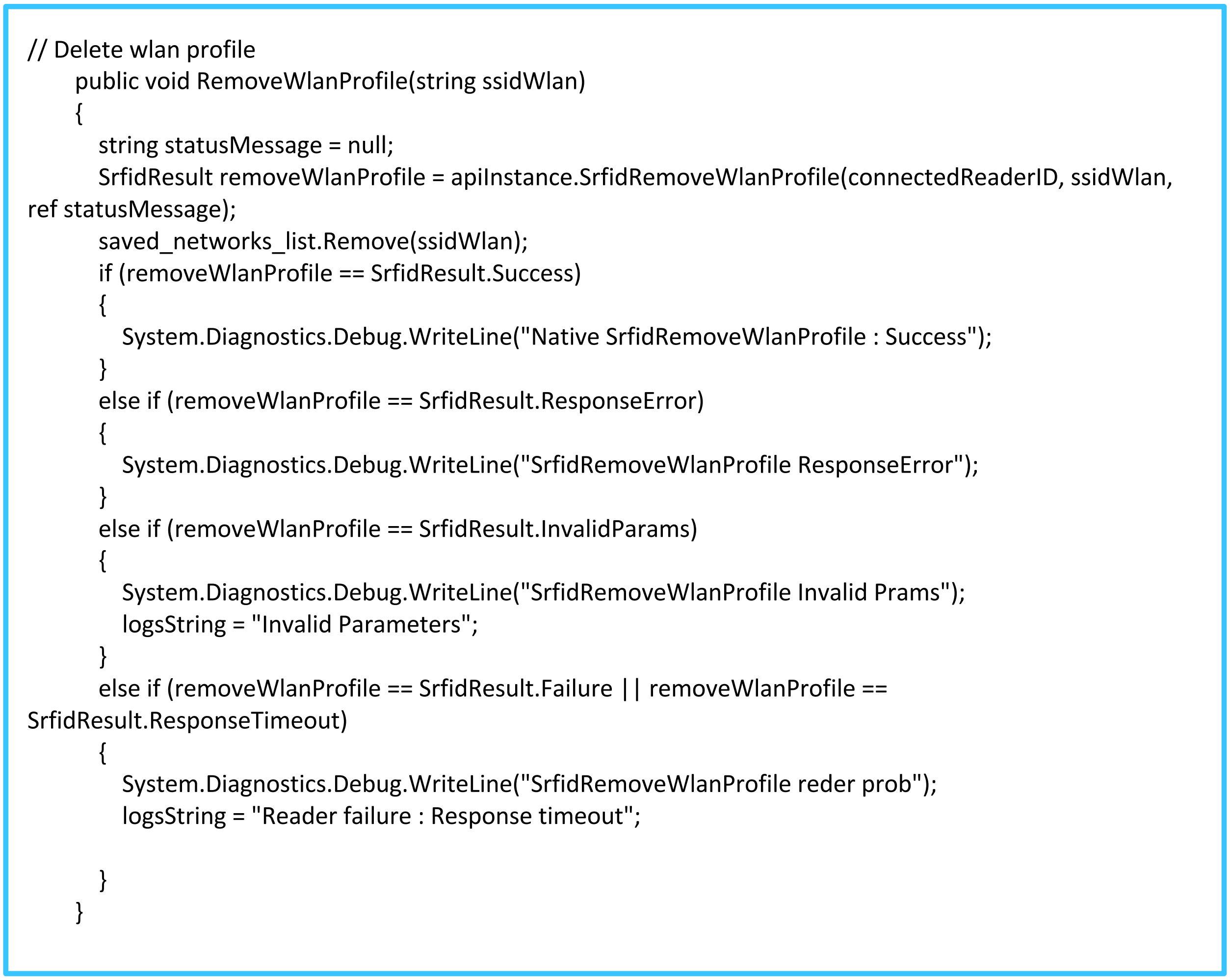
```
public SrfidResult AddWlanProfile(sRfidAddProfileConfig profileConfig)
    {
      string statusMessage = null;
      SrfidResult addWlanProfile = apiInstance.SrfidAddWlanProfile(connectedReaderID, profileConfig, ref statusMessage);

      if (addWlanProfile == SrfidResult.Success)
      {
        System.Diagnostics.Debug.WriteLine("Native SrfidAddWlanProfile : Success");
        logsString = "Success";
      }
      else if (addWlanProfile == SrfidResult.ResponseError)
      {
        System.Diagnostics.Debug.WriteLine("SrfidAddWlanProfile ResponseError");
        logsString = "Response Error";
      }
      else if (addWlanProfile == SrfidResult.InvalidParams)
      {
        System.Diagnostics.Debug.WriteLine("SrfidAddWlanProfile Invalid Prams");
        logsString = "Invalid Parameters";
      }
      else if (addWlanProfile == SrfidResult.Failure || addWlanProfile == SrfidResult.ResponseTimeout)
      {
        System.Diagnostics.Debug.WriteLine("SrfidAddWlanProfile reder prob");
        logsString = "Reader failure : Response timeout";
      }
      return addWlanProfile;
    }
```

Save WLAN Profile

```
public SrfidResult SaveWlanProfile()
    {
      string statusMessage = null;
      SrfidResult saveWlanProfile = apiInstance.SrfidWlanSaveProfile(connectedReaderID, ref
statusMessage);
      if (saveWlanProfile == SrfidResult.Success)
      {
        System.Diagnostics.Debug.WriteLine("Native SrfidWlanSaveProfile : Success");
      }
      else if (saveWlanProfile == SrfidResult.ResponseError)
      {
        System.Diagnostics.Debug.WriteLine("SrfidWlanSaveProfile ResponseError");
        logsString = "Response Error";
      }
      else if (saveWlanProfile == SrfidResult.InvalidParams)
      {
        System.Diagnostics.Debug.WriteLine("SrfidWlanSaveProfile Invalid Prams");
        logsString = "Invalid Parameters";
      }
      else if (saveWlanProfile == SrfidResult.Failure || saveWlanProfile == SrfidResult.ResponseTimeout)
      {
        System.Diagnostics.Debug.WriteLine("SrfidWlanSaveProfile reder prob");
        logsString = "Reader failure : Response timeout";
      }
      return saveWlanProfile;
    }
```

Remove WLAN Profile



```
// Delete wlan profile
       public void RemoveWlanProfile(string ssidWlan)
       {
           string statusMessage = null;
           SrfidResult removeWlanProfile = apiInstance.SrfidRemoveWlanProfile(connectedReaderID, ssidWlan,
ref statusMessage);
           saved_networks_list.Remove(ssidWlan);
           if (removeWlanProfile == SrfidResult.Success)
           {
              System.Diagnostics.Debug.WriteLine("Native SrfidRemoveWlanProfile : Success");
           }
           else if (removeWlanProfile == SrfidResult.ResponseError)
           {
              System.Diagnostics.Debug.WriteLine("SrfidRemoveWlanProfile ResponseError");
           }
           else if (removeWlanProfile == SrfidResult.InvalidParams)
           {
              System.Diagnostics.Debug.WriteLine("SrfidRemoveWlanProfile Invalid Prams");
              logsString = "Invalid Parameters";
           }
           else if (removeWlanProfile == SrfidResult.Failure || removeWlanProfile ==
SrfidResult.ResponseTimeout)
           {
              System.Diagnostics.Debug.WriteLine("SrfidRemoveWlanProfile reder prob");
              logsString = "Reader failure : Response timeout";

           }
       }
```

Connect WLAN Profile

```
// Connect wlan profile
    public SrfidResult ConnectWlanProfile(string ssid)
    {
      string statusMessage = null;
      SrfidResult connectWlanProfile = SrfidResult.Failure;
      for (int i = 0; i < 2; i++)
      {
        connectWlanProfile = apiInstance.SrfidconnectWlanProfile(connectedReaderID, ssid, ref
statusMessage);
        if (connectWlanProfile == SrfidResult.Success)
        {
          System.Diagnostics.Debug.WriteLine("Native conectWlanProfile : Success");
        }
        else if (connectWlanProfile == SrfidResult.ResponseError)
        {
          System.Diagnostics.Debug.WriteLine("conectWlanProfile ResponseError");
        }
        else if (connectWlanProfile == SrfidResult.InvalidParams)
        {
          System.Diagnostics.Debug.WriteLine("conectWlanProfile Invalid Prams");
          logsString = "Invalid Parameters";
        }
        else if (connectWlanProfile == SrfidResult.Failure || connectWlanProfile ==
SrfidResult.ResponseTimeout)
        {
          System.Diagnostics.Debug.WriteLine("conectWlanProfile reder prob");
          logsString = "Reader failure : Response timeout";
        }
      }

      return connectWlanProfile;
    }
```

Get WLAN Certificates List

```
  public SrfidResult GetWlanCertificatesList(NSMutableArray wlanCertificatesList)
        {
            string statusMessage = null;
            IntPtr availableHandle = wlanCertificatesList.Handle;
            SrfidResult getWlanCertificatesListApiCall =
apiInstance.SrfidGetWlanCertificatesList(connectedReaderID, ref availableHandle, ref statusMessage);

            if (getWlanCertificatesListApiCall == SrfidResult.Success)
            {
                System.Diagnostics.Debug.WriteLine("SrfidGetWlanCertificatesList Success");
                logsString = "Success";
            }
            else if (getWlanCertificatesListApiCall == SrfidResult.ResponseError)
            {
                System.Diagnostics.Debug.WriteLine("SrfidGetWlanCertificatesList ResponseError");
                logsString = "Response Error";

            }
            else if (getWlanCertificatesListApiCall == SrfidResult.Failure || getWlanCertificatesListApiCall ==
SrfidResult.ResponseTimeout)
            {
                System.Diagnostics.Debug.WriteLine("SrfidGetWlanCertificatesList Failure");
                logsString = "Failure";
            }

            return getWlanCertificatesListApiCall;


        }
```

Disconnect WLAN Profile

```
    // WLAN disconnect
        public SrfidResult DisconnectWlanProfile()
        {
            string statusMessage = null;
            SrfidResult disconnectWlanProfile = apiInstance.SrfidWlanDisConnectProfile(connectedReaderID, ref
statusMessage);

            if (disconnectWlanProfile == SrfidResult.Success)
            {
                System.Diagnostics.Debug.WriteLine("Native disconnectWlanProfile : Success");
            }
            else if (disconnectWlanProfile == SrfidResult.ResponseError)
            {
                System.Diagnostics.Debug.WriteLine("disconnectWlanProfile ResponseError");

            }
            else if (disconnectWlanProfile == SrfidResult.InvalidParams)
            {
                System.Diagnostics.Debug.WriteLine("disconnectWlanProfile Invalid Prams");
                logsString = "Invalid Parameters";
            }
            else if (disconnectWlanProfile == SrfidResult.Failure || disconnectWlanProfile ==
SrfidResult.ResponseTimeout)
            {
                System.Diagnostics.Debug.WriteLine("disconnectWlanProfile reder prob");
                logsString = "Reader failure : Response timeout";
            }

            return disconnectWlanProfile;
        }
```

Known Issues

- There is an issue in the MAUI Wrapper when getting supported channels for a specific region that the Reader supports. Therefore, it is unable to set a value for *EnableChannels* property when setting a new Regulatory Configuration to a Reader.
- API for getting Link Profiles for Antenna configuration is not implemented in MAUI Wrapper. Therefore, supported Link profile names are hardcoded and show those as a list for user selection and not able to validate *Tari* value when saving Antenna configuration.

Appendix

Link profile values can be found below

- 60000 MV 4 1500 25000 25000 0
- 640000 MV FMO 1500 6250 6250 0
- 640000 MV FMO 2000 6250 6250 0
- 120000 MV 2 1500 25000 25000 0
- 120000 MV 2 1500 12500 23000 2100
- 120000 MV 2 2000 25000 25000 0
- 120000 MV 2 2000 12500 23000 2100
- 128000 MV 2 1500 25000 25000 0
- 128000 MV 2 1500 12500 23000 2100
- 128000 MV 2 2000 25000 25000 0
- 128000 MV 2 2000 12500 23000 2100
- 160000 MV 2 1500 12500 18800 2100
- 160000 MV 2 2000 12500 18800 2100
- 60000 MV 4 1500 12500 23000 2100
- 60000 MV 4 2000 25000 25000 0
- 60000 MV 4 2000 12500 23000 2100
- 64000 MV 4 1500 25000 25000 0
- 64000 MV 4 1500 12500 23000 2100
- 64000 MV 4 2000 25000 25000 0
- 64000 MV 4 2000 12500 23000 2100
- 80000 MV 4 1500 12500 18800 2100
- 80000 MV 4 2000 12500 18800 2100
- 668 MV FMO 668 668 668 668
- 320000 MV FMO 1500 12500 18800 2100
- 320000 MV FMO 2000 12500 18800 2100

- 30000 MV 8 1500 25000 25000 0
- 30000 MV 8 1500 12500 23000 2100
- 30000 MV 8 2000 25000 25000 0
- 30000 MV 8 2000 12500 23000 2100
- 32000 MV 8 1500 25000 25000 0
- 32000 MV 8 1500 12500 23000 2100
- 32000 MV 8 2000 25000 25000 0
- 32000 MV 8 2000 12500 23000 2100
- 40000 MV 8 1500 12500 18800 2100
- 40000 MV 8 2000 12500 18800 2100