ZEBRA RFID SDK FOR ANDROID

Developer Guide

ZEBRA

MN-003158-05 Rev. A

Copyright

ZEBRA and the stylized Zebra head are trademarks of Zebra Technologies Corporation, registered in many jurisdictions worldwide. All other trademarks are the property of their respective owners. ©2019 Zebra Technologies Corporation and/or its affiliates. All rights reserved.

COPYRIGHTS & TRADEMARKS: For complete copyright and trademark information, go to www.zebra.com/copyright.

WARRANTY: For complete warranty information, go to www.zebra.com/warranty.

END USER LICENSE AGREEMENT: For complete EULA information, go to www.zebra.com/eula.

For Australia Only

For Australia Only. This warranty is given by Zebra Technologies Asia Pacific Pte. Ltd., 71 Robinson Road, #05-02/03, Singapore 068895, Singapore. Our goods come with guarantees that cannot be excluded under the Australia Consumer Law. You are entitled to a replacement or refund for a major failure and compensation for any other reasonably foreseeable loss or damage. You are also entitled to have the goods repaired or replaced if the goods fail to be of acceptable quality and the failure does not amount to a major failure.

Zebra Technologies Corporation Australia's limited warranty above is in addition to any rights and remedies you may have under the Australian Consumer Law. If you have any queries, please call Zebra Technologies Corporation at +65 6858 0722. You may also visit our website: www.zebra.com for the most updated warranty terms.

Terms of Use

- Proprietary Statement

  This manual contains proprietary information of Zebra Technologies Corporation and its subsidiaries ("Zebra Technologies"). It is intended solely for the information and use of parties operating and maintaining the equipment described herein. Such proprietary information may not be used, reproduced, or disclosed to any other parties for any other purpose without the express, written permission of Zebra Technologies.
- Product Improvements

  Continuous improvement of products is a policy of Zebra Technologies. All specifications and designs are subject to change without notice.
- Liability Disclaimer

  Zebra Technologies takes steps to ensure that its published Engineering specifications and manuals are correct; however, errors do occur. Zebra Technologies reserves the right to correct any such errors and disclaims liability resulting therefrom.
- Limitation of Liability

  In no event shall Zebra Technologies or anyone else involved in the creation, production, or delivery of the accompanying product (including hardware and software) be liable for any damages whatsoever (including, without limitation, consequential damages including loss of business profits, business interruption, or loss of business information) arising out of the use of, the results of use of, or inability to use such product, even if Zebra Technologies has been advised of the possibility of such damages. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you.

Revision History

Changes to the original manual are listed below:

Change	Date	Description
-05 Rev A	12/2019	Copyright update. Updated text and screen shots for 123RFID Mobile Application.
-04 Rev A	03/2019	Formatting updates.
-03 Rev A	10/2018	Added SDK for Xamarin Android.
-02 Rev A	05/2018	Updated: - Demo Application Source folder - Application Gradle Modification for Module screen - Connection Management - changed ".mot" to ".zebra" - Write, Block-Write; Block-Erase. - URLs to conform with Tech Pubs style - Page breaks Added: - Set Default Configuration. - RFD8500 information.
-01 Rev A	12/2017	Initial release

Table of Contents

Copyright 2
For Australia Only 2
Terms of Use 2
Revision History 3

About
Introduction 7
Supported RFID Readers 7
Chapter Descriptions 7
Notational Conventions 8
Related Documents and Software 8
Service Information 9
Provide Documentation Feedback 9

Getting Started
Introduction 10
Installing Android Studio 10
Importing the 123RFID Mobile Application Project 11
Building and Running Projects 13
RFID API3 Android SDK 16
Creating an Android Project 16
Installing Xamarin 18

ZEBRA RFID SDK for Android
Introduction 22
Basics 22
Connection Management 23
Connect to an RFID Reader 23
Disconnect 25
Reconnect (RFD8500 only) 25
Dispose 25
Knowing the Reader Capabilities 26
General Capabilities 26
Regulatory Capabilities 26
UHF Band Capabilities 26

Reader Identification ... 27
Configuring the Reader ... 28
RF Mode ... 28
Antenna Specific Configuration ... 28
Reset Configuration to Factory Defaults ... 31
Managing Events ... 31
Basic Operations ... 34
Tag Storage Settings ... 34
Advanced Operations ... 39
Using Pre-Filters ... 39
Using Triggers ... 41
Access ... 42
Using Access Sequence ... 44
Gen2v2 Operations ... 45
Resetting the Reader ... 47
Tag Locationing ... 47
Trigger Mode - RFID and Barcode ... 48
Set Attribute ... 50
Set Host LED Support ... 50
Set Default Configuration ... 50
Exceptions ... 51
Exception Handling ... 52
General Guidelines ... 52
Synchronization ... 52
Threading ... 52
Quick Start Sample ... 53

ZEBRA RFID SDK for Xamarin Android

Introduction ... 57
Basics ... 57
Connection Management ... 58
Connect to an RFID Reader ... 58
Disconnect ... 59
Dispose ... 59
Knowing the Reader Capabilities ... 60
General Capabilities ... 60
Regulatory Capabilities ... 60
UHF Band Capabilities ... 60
Reader Identification ... 61
Configuring the Reader ... 62
RF Mode ... 62
Antenna Specific Configuration ... 62
Reset Configuration to Factory Defaults ... 65
Managing Events ... 65
Basic Operations ... 68
Tag Storage Settings ... 68
Advanced Operations ... 74
Using Pre-Filters ... 74
Introduction ... 74
Trigger Mode - RFID and Barcode ... 85

Set Attribute 88
Set Host LED Support 88
Set Default Configuration 88
Exceptions 89
Exception Handling 89
General Guidelines 90
Synchronization 90
Threading 90
Quick Start Sample 91

Migrating to a Combined RFD8500/RFD2000 RFID SDK
Introduction 96
Using Existing Applications With The RFID SDK 96
Migrating and Supporting RFD2000 Applications 97
Summarizing Application Support for RFD8500 and RFD2000 Readers 98
Examples 98

About

Introduction

The Zebra RFID SDK for Android Developer Guide provides installation and programming information for the Software Developer Kit (SDK) that allows RFID application development for the Zebra Android devices.

NOTE: This guide provides details and references to SDK using both Java and Xamarin.

Please refer to respective sections and snippets based on application type:

- Android Application development using Android Studio IDE and Java programming language.
- Android Application development using Xamarin - Visual Studio IDE and C# programming language.

Supported RFID Readers

The following RFID Readers are supported:

- RFD2000
- RFD8500
- MC3300R

Chapter Descriptions

Topics covered in this guide are as follows:

- *Getting Started* provides an overview of the RFID SDK and sample mobile application usage, build, and operation.
- *ZEBRA RFID SDK for Android* provides detailed information about how to use various basic and advanced functionality to develop an Android application using the Zebra RFID SDK for Android.
- *ZEBRA RFID SDK for Xamarin Android* provides detailed information about how to use various basic and advanced functionality to develop an Xamarin Android application using the Zebra RFID SDK for Xamarin Android.
- *Migrating to a Combined RFD8500/RFD2000 RFID SDK* provides the information necessary to update existing applications and develop new applications for use with the combined RFID SDK.

Notational Conventions

The following conventions are used in this document:

- **Bold** text is used to highlight the following:
  - Key names on a keypad
  - Button names on a screen
- Bullets (•) indicate:
  - Action items
  - Lists of alternatives
  - Lists of required steps that are not necessarily sequential
- Sequential lists (e.g., those that describe step-by-step procedures) appear as numbered lists.

Related Documents and Software

The following documents provide more information about the readers.

- RFD2000 User Guide, p/n MN-003128-xx.
- RFD2000 Quick Start Guide, p/n MN-003129-xx.
- TC20 Quick Start Guide, p/n MN-003018-xx.
- TC20 User Guide, p/n MN-003020-xx.
- Zebra Scanner SDK for Android Developer Guide, p/n MN002223AXX.
- RFD8500 RFID Developer Guide, p/n MN002222AXX.
- RFD8500 Quick Start Guide, p/n MN002225AXX.
- RFD8500 Regulatory Guide, p/n MN002062AXX.
- MC3300R User Guied, p/n MN-003180-XX.
- MC40 User Guide, p/n MN000111AXX.
- TC55 User Guide, p/n MN000015AXX.
- TC70 User Guide, p/n MN-002890-XX.
- Java Class Reference Guide - This guide is in HTML format located under the javadoc directory in the RFID SDK for Android distribution package.
- RFD8500 User Guide, p/n MN002065Axx.
- RFD8500i User Guide, p/n MN-002761-XX.
- RFD8500i Quick Start Guide, p/n MN-002760-XX
- RFD8500i Regulatory Guide, p/n MN-002856-xx.
- RFD8500 Bluetooth Pairing Using S/N Barcode White Paper, available at: www.zebra.com/support.
- Zebra RFD8500 Attribute Data Dictionary, available at: www.zebra.com/support.
- Zebra Scanner SDK Attribute Data Dictionary. p/n 72E-149786-XX.

For the latest version of this guide and all guides, go to: www.zebra.com/support.

Service Information

If you have a problem using the equipment, contact your facility's technical or systems support. If there is a problem with the equipment, they will contact the Zebra Global Customer Support Center at: www.zebra.com/support.

When contacting Zebra support, please have the following information available:

- Serial number of the unit
- Model number or product name
- Software type and version number.

Zebra responds to calls by e-mail, telephone or fax within the time limits set forth in support agreements.

If your problem cannot be solved by Zebra support, you may need to return your equipment for servicing and will be given specific directions. Zebra is not responsible for any damages incurred during shipment if the approved shipping container is not used. Shipping the units improperly can possibly void the warranty.

If you purchased your business product from a Zebra business partner, contact that business partner for support.

Provide Documentation Feedback

If you have comments, questions, or suggestions about this guide, send an email to EVM-Techdocs@zebra.com.

Getting Started

Introduction

This chapter provides instruction on importing and running the 123RFID Mobile Application code and instructions.

Installing Android Studio

To install Android Studio go to https://developer.android.com/studio/ and click DOWNLOAD ANDROID STUDIO.

The project uses the following configurations:

- Minimum SDK Version - API 19: Android 4.4 (KitKat)
- Target SDK Version - API 25: Android 7.1.1 (Nougat)
- Gradle Version – 3.3 or latest matching with Android Studio version.
- Java Version - Java7
- Android Studio version 2.3 and onwards.

NOTE: RFID API3 Android SDK depends upon android.support-v4 or v7 to run. If the Android application is created without 'appcompat' support, add 'com.android.support:support-v4' or v7 in the gradle file 'dependencies' section. If the application is created with the application compatibility option, dependencies are automatically added.

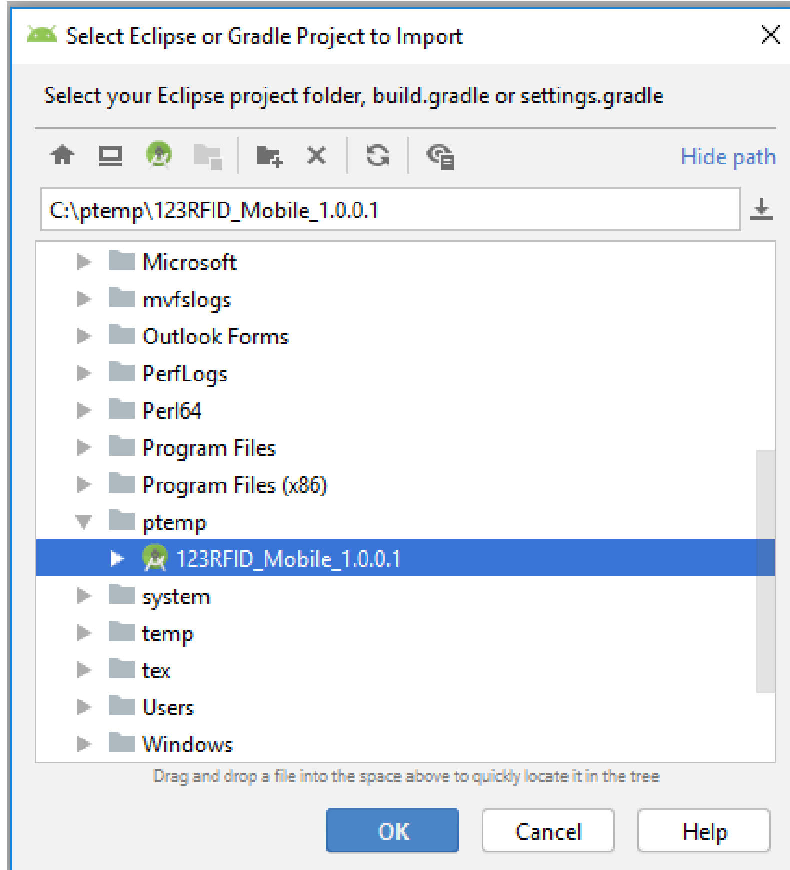
The latest SDK tools packaged with Android Studio are acceptable. To use the SDK manager to download the required Android SDK packages go to **Menu Tools** > **Android** > **SDK Manager**.

Importing the 123RFID Mobile Application Project

To import the demo application project:

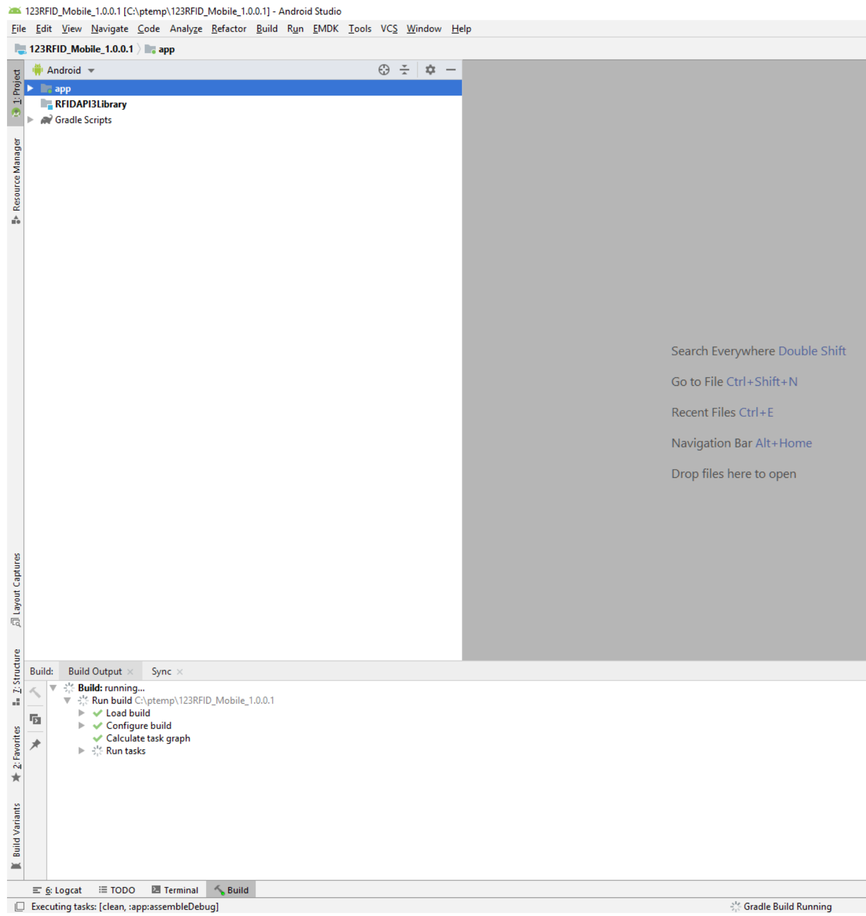
1. Open **Android Studio**. The Android Studio screen displays.
2. Select **Import Project** to set the language and the SDK tool path.
3. Select **123RFID Mobile API** (from Demo application source folder).

Figure 1 Project Folder



4. Android Studio automatically synchronizes the SDK path if required.

Figure 2 Syncing Android SDKs

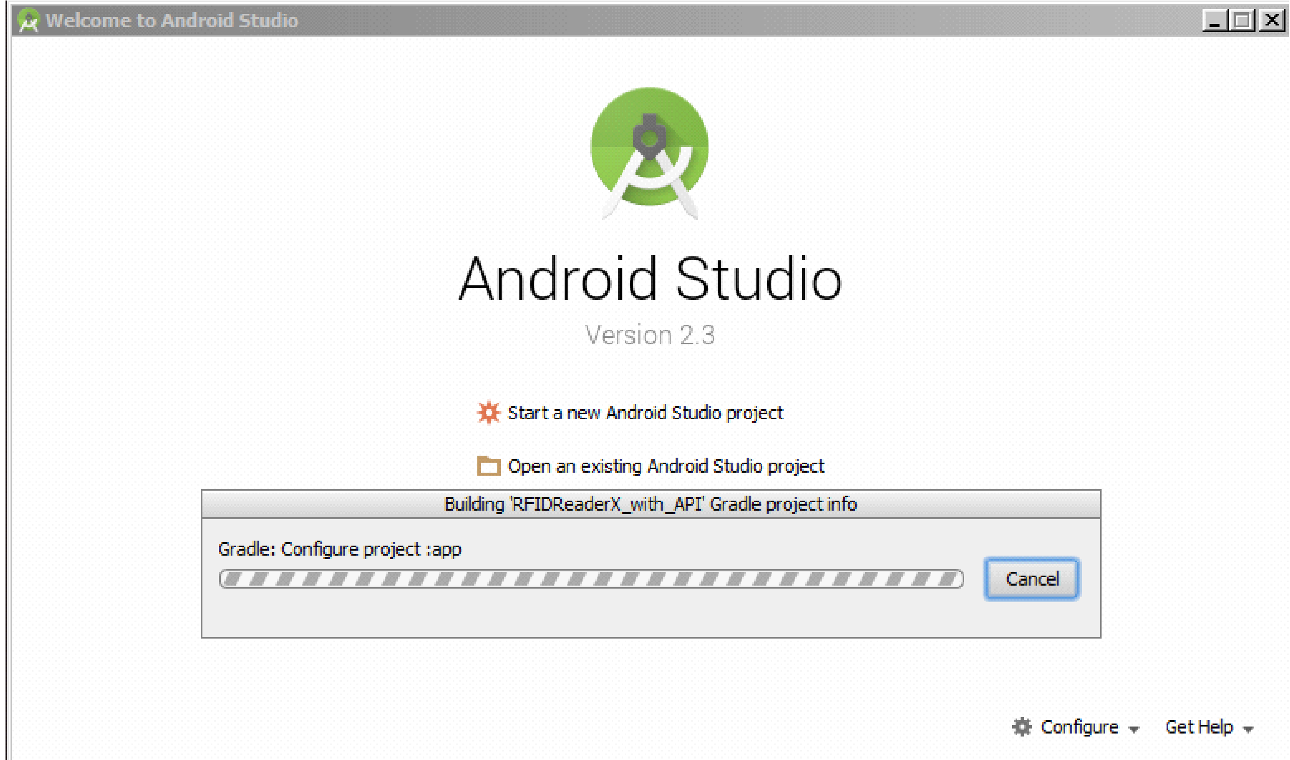


Building and Running Projects

To build and run a project:

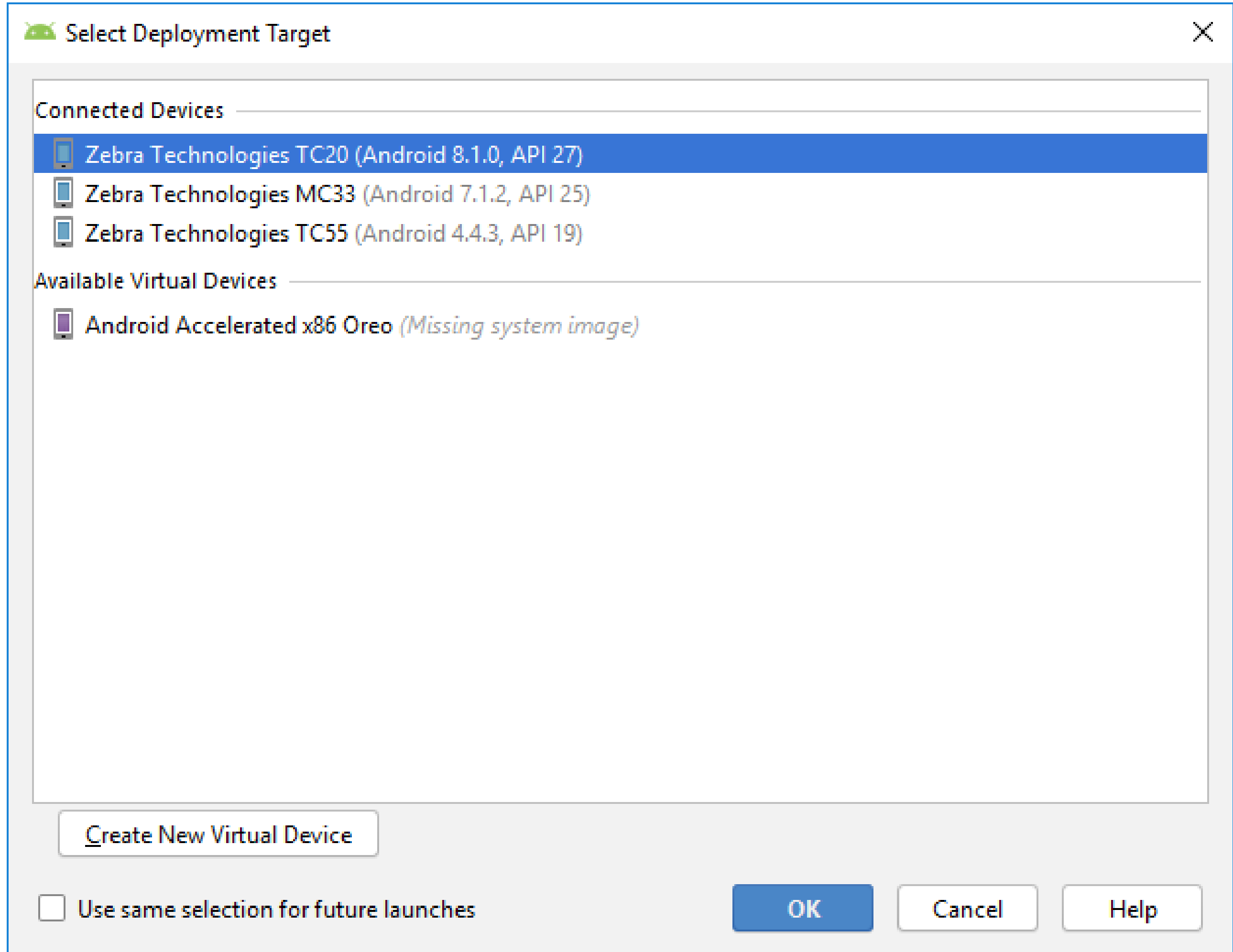
1. Android Studio may start downloading required Gradle/SDK packages first. Define proxy information in Android Studio and gradle properties as required and Android Studio starts to build the project.

Figure 3 Building the Project



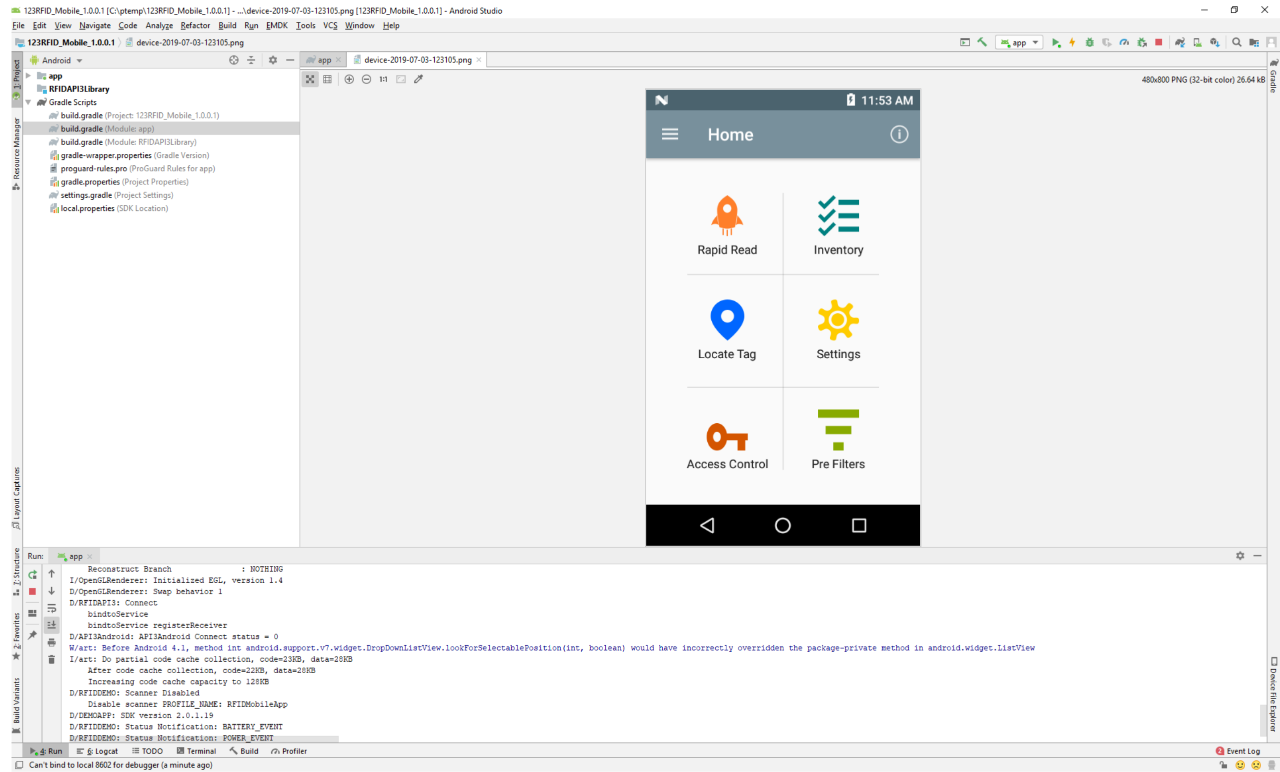
2. After completion of a successful compilation, run the application using the **Run App** button. Android studio prompts for the deployment target. Install the built application on the required device by choosing a device from the **Connected Devices** list.

Figure 4 Choose Connected Device Screen



3. Figure 5 is the captured image of the demo application running on an Android device.

Figure 5 Demo Application Screen



RFID API3 Android SDK

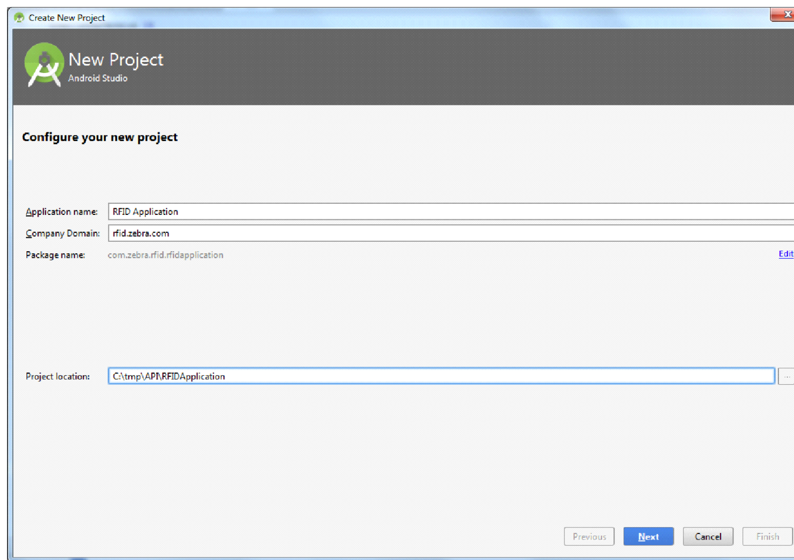
RFID API3 SDK for Android is provided in 'aar' package format. Create RFIDAPI3Library folder and copy the API3_LIB-release-X.X.X.X.aar inside the folder at a known local path.

Creating an Android Project

To create an Android Project:

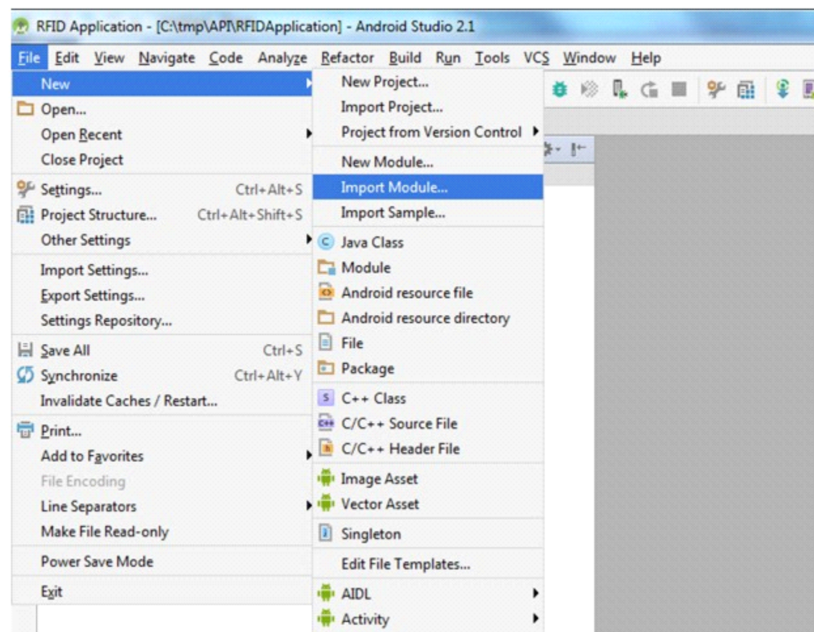
1. Select **File** > **New** > **New Project** to create a new Android project and follow the on screen steps in the Android Studio New Project wizard.

Figure 6 Create New Project Screen



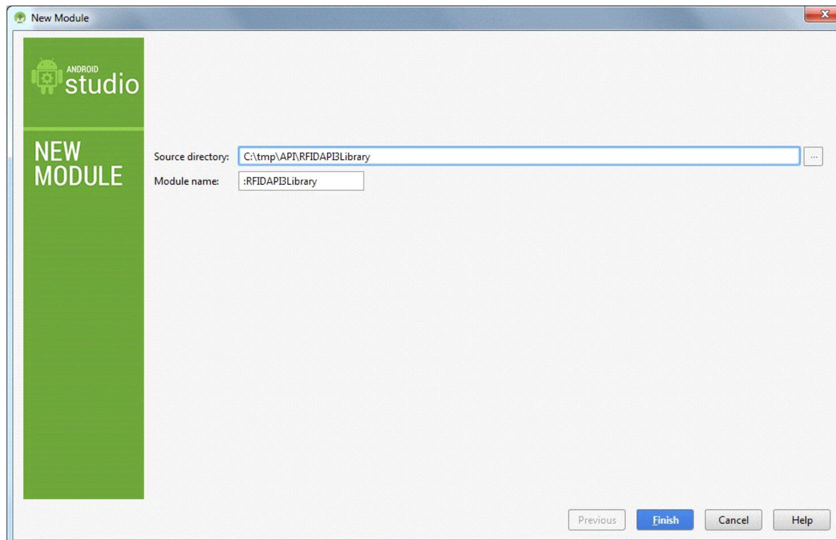
2. Navigate to **File** > **New** > **Import Module** to import the RFID API3 module.

Figure 7 Import RFID API3 Module Screen



3. Browse the **RFIDAPI3Library** folder (after source directory selection, the module name displays as **RFIDAPI3Library**).

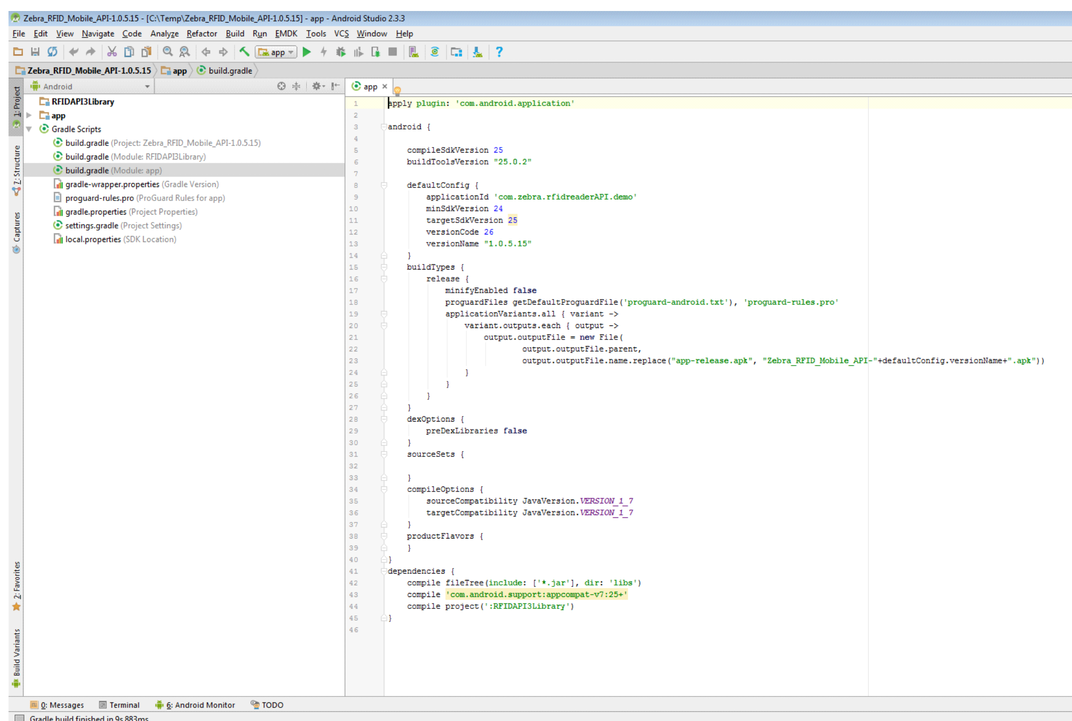
Figure 8 New Module Screen



4. Add the module under dependencies in the application **gradle** file. The dependencies section shows 'appcompat' and RFIDAPI3 module as follows:

```
dependencies {
    //.... other dependencies
    implementation 'com.android.support:appcompat-v7:28.0.0'
    implementation project(':RFIDAPI3Library')
    //....
}
```

Figure 9 Application Gradle Modification for Module



5. The application is ready to import the com.zebra.rfid.api3.* package/class.

Figure 10 Application MainAcitivity.java with Imported RFID API3 Package

```
package com.zebra.rfid.rfidapplication;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;

import com.zebra.rfid.api3.RFIDReader;

public class MainActivity extends AppCompatActivity {

    public static RFIDReader mConnectedReader;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
        fab.setOnClickListener((view) → {
                Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
                        .setAction("Action", null).show();
        });

    }
```

Installing Xamarin

To install Xamarin Android, click on Xamarin and follow the procedure. The guide and project uses the following configurations:

- Minimum SDK Version - API 19: Android 4.4 (KitKat)
- Target SDK Version - API 25: Android 7.1.1 (Nougat)
- .NET Framework 4.6.1 Development environment
- Visual Studio 2017
- Xamarin Version: Xamarin.Android 7.0

NOTE: RFID API3 Xamarin SDK depends upon android AppCompatSupport library. AppCompatSupportV4 or V7 library is mandatory for integrating the dll into the application. Applications created with standard templates have Xamarin.Android.Support.v4 and Xamarin.Android.Support.v7.AppCompat automatically added as part of assembly reference in the Solution project.

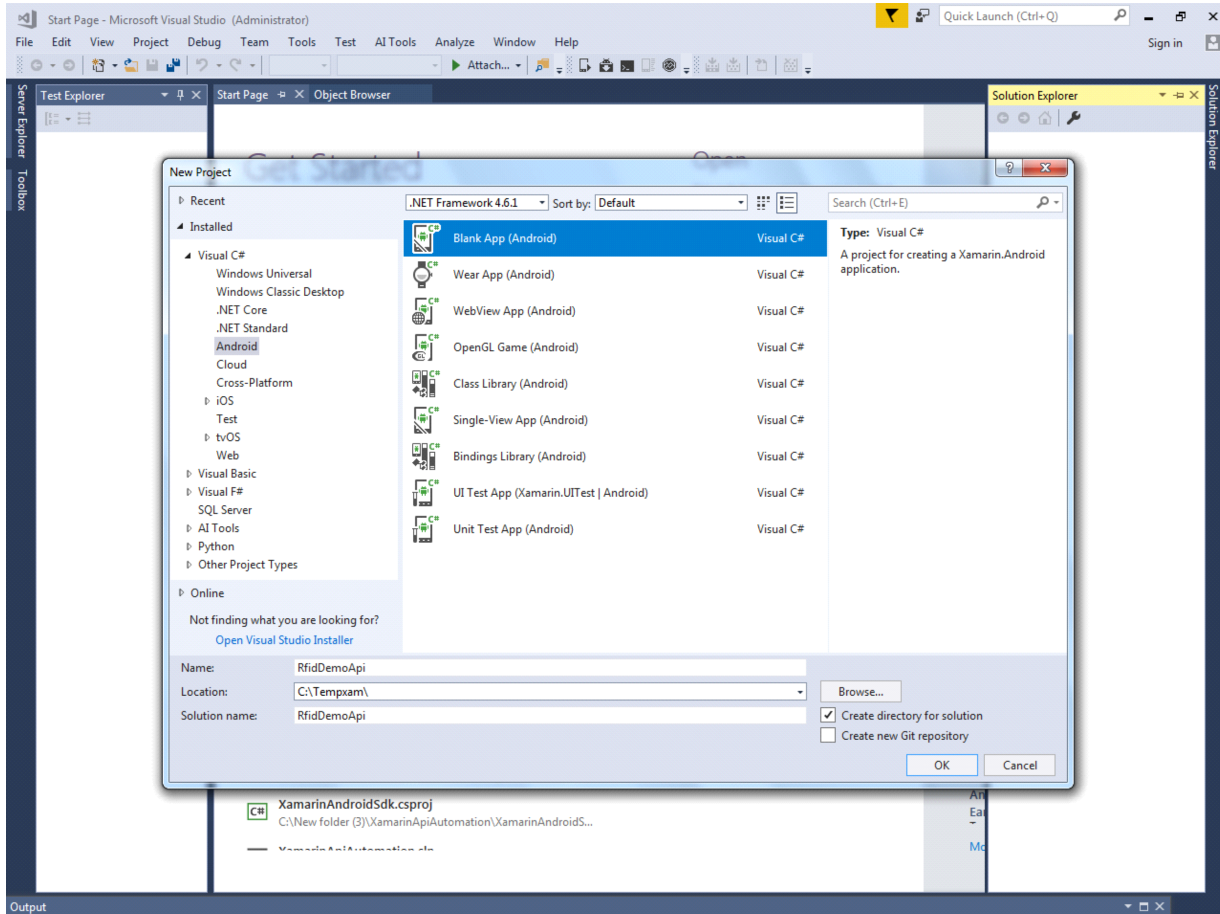
Creating a Xamarin Project

To create a C# Android Project in Visual Studio 2017:

1. Start Visual Studio 2017 and select **File** > **New** > **Project** >**Visual C#** >**Android**.
2. Create a Blank App (Android) project and follow the on-screen steps in Visual Studio.
3. Create a RfidSdk folder inside the project.
4. Copy XamarinZebraRFID.dll in RfidSdk folder.

5. Add a XamarinZebraRFID.dll reference in to the project reference from the RfidSdk folder.

Figure 11 Create New Project



6. Import Com.Zebra.Rfid.Api3 namespace/classes.

Figure 12 Adding XamarinZebraRFID.dll to References

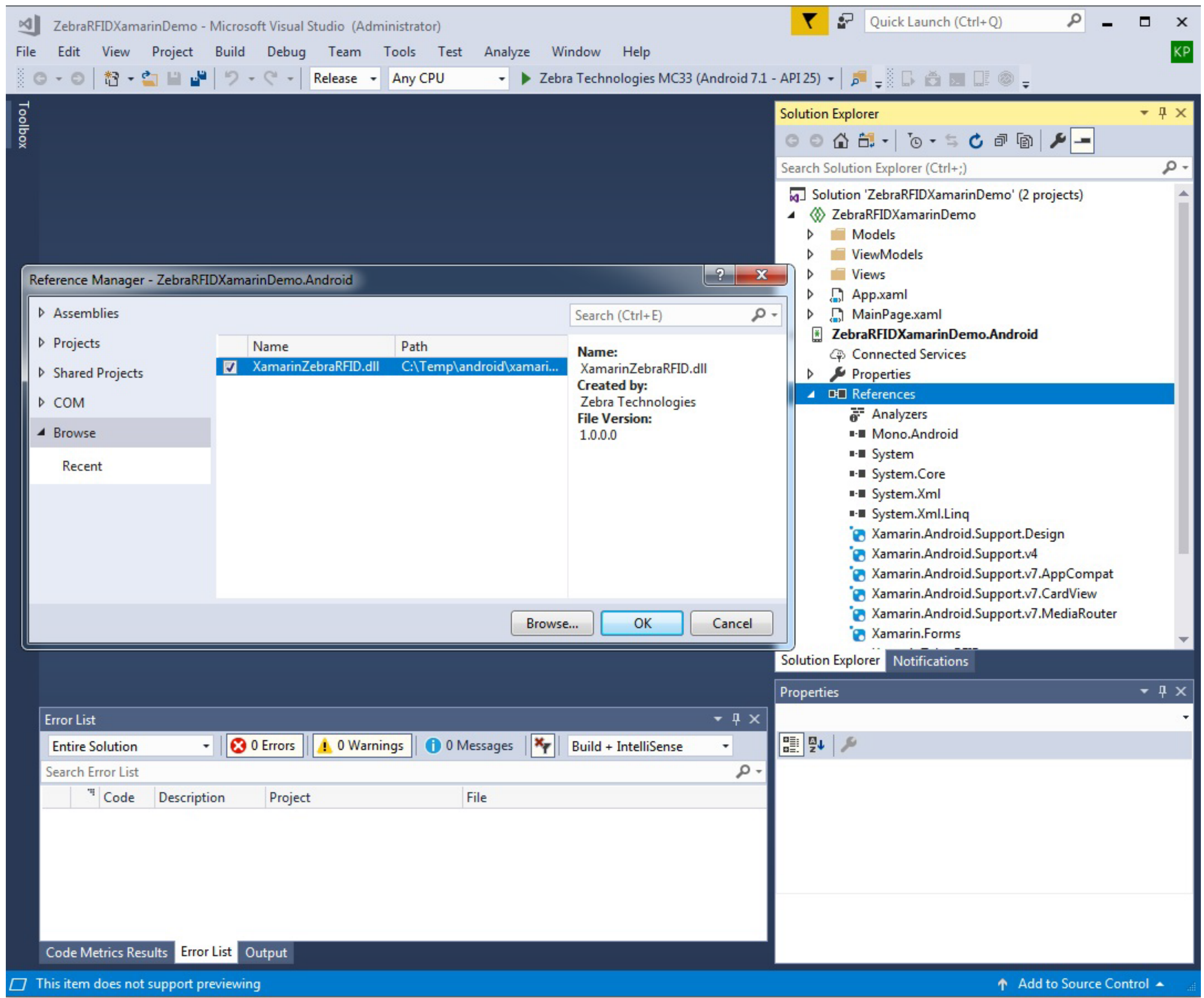
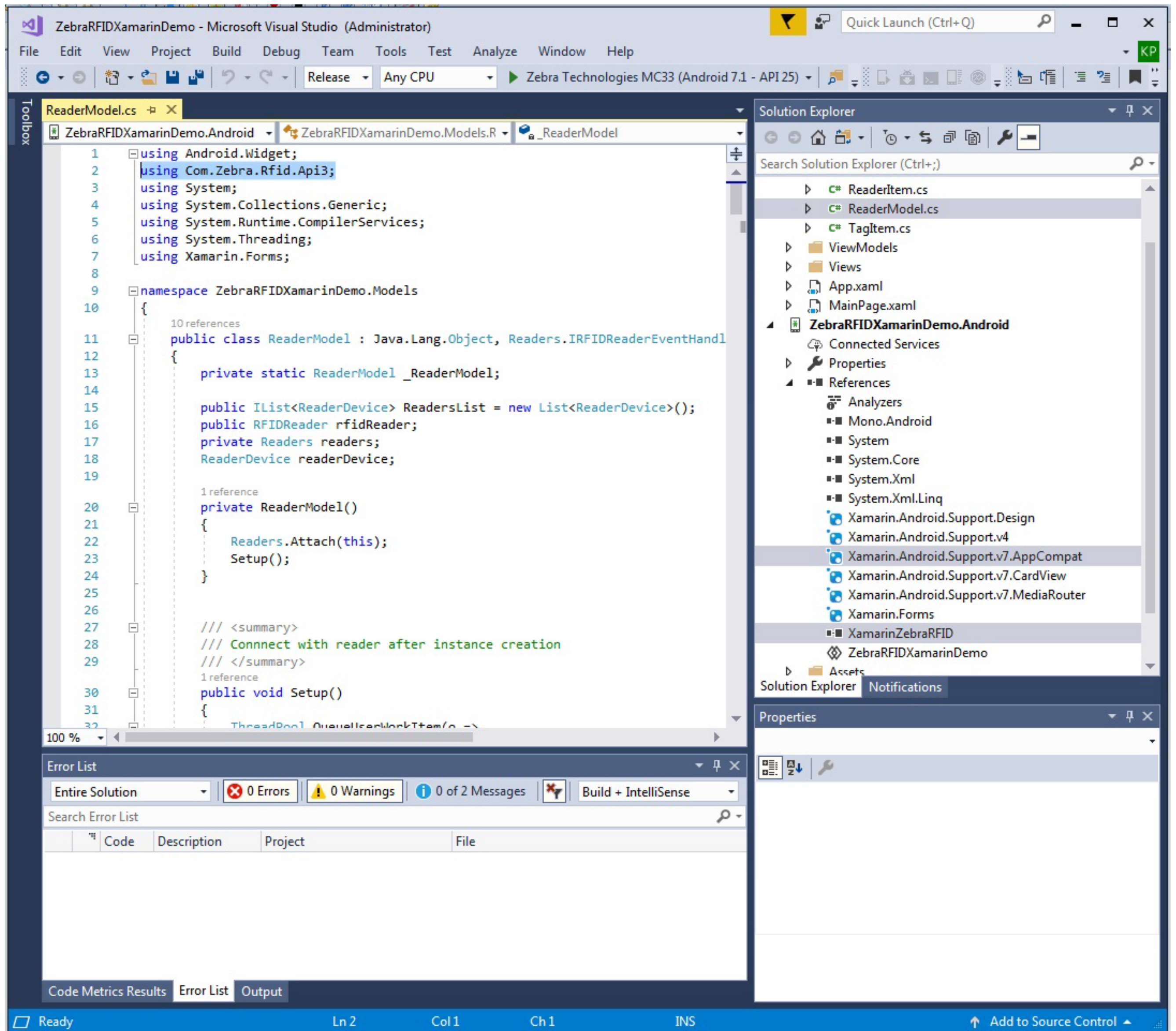


Figure 13 Example Application with RFID SDK



ZEBRA RFID SDK for Android

Introduction

This chapter provides detailed information about how to use various basic and advanced functionality to develop an Android application using the Zebra RFID SDK for Android.

The Zebra RFID SDK for Android allows applications to communicate with RFID readers connected to a mobile device.

The Zebra RFID SDK for Android provides the API that can be used by external applications to manage connection of the RFID readers, and to control connected RFID reader.

Basics

NOTE: Detailed API documentation can be found in Java Class Reference Guide distributed with the Zebra RFID SDK for Android (see Related Documents and Software on page 8).

The Zebra RFID SDK for Android provides the ability to manage RFID readers' connections, perform various operations with connected RFID readers, configure connected RFID readers and knowing other information related to connected RFID readers.

Zebra RFID Android SDK consists of a static android library in 'aar' format that is supposed to be linked with an external Android application. Step-by-step instructions for configuring the Android application project to enable utilization of Zebra RFID Android SDK are provided in *Getting Started: Installing Android Studio on page 10, Building and Running Projects on page 13,* and RFID API3 Android SDK on page 16.

All available APIs are defined under the com.zebra.rfid.api3 package. RFIDReader is the root Java class in the SDK. The application uses a single instance of an RFIDReader object to interact with a particular reader.

Use available readers and the RFIDReader object to register for events, connect with readers, and after successful connection, perform required operations such as inventory, access and locate.

It is recommended that all API calls should be made using Android 'AsyncTask' so that operations are performed in the background thread keeping the UI thread free.

If the API call fails, the API throws an exception. The application should call all APIs in try-catch block for handling exceptions.

Connection Management

Connect to an RFID Reader

Connection is the first step to talk to an RFID reader. Importing package is the first step to use any API. Import the package as follows:

```
import com.zebra.rfid.api3.*;
```

As first step create instance of Readers class with passing activity context as first parameter and enum value SERVICE_SERIAL as second parameter

```
readers = new Readers(this, ENUM_TRANSPORT.SERVICE_SERIAL);
```

The Readers class instance gives a list of all available/paired RFID readers with an Android device. Readers list is in the form of ReaderDevice class.

```
ArrayList<ReaderDevice> readersListArray = readers.GetAvailableRFIDReaderList();
ReaderDevice readerDevice = availableRFIDReaderList.get(0);
```

ReaderDevice class includes instance of RFIDReader class; which is root class to interface and performing all operations with RFID reader.

```
// Establish connection to the RFID Reader
rfidReader.connect();
```

In addition, the application can implement Readers.RFIDReaderEventHandler in the following way to get notified for RFID reader getting attached (added) / detached (removal).

```
public class MainActivity extends ActionBarActivity implements
        Readers.RFIDReaderEventHandler {
    @Override
    public void RFIDReaderAppeared(ReaderDevice device) {
        // handle reader addition
    }
    @Override
    public void RFIDReaderDisappeared(ReaderDevice device) {
        // handle reader removal
    }
}
```

Special Connection Handling Cases

In a normal scenario, the reader connects fine, but the following are the cases which require special handling at the time of connection.

The following example demonstrates the connection is handled under try-catch block and OperationFailure exception is thrown by connection API is stored and used for further analysis.

```
private OperationFailureException ex;
        try {
              // Establish connection to the RFID Reader
            rfidReader.connect();
        } catch (InvalidUsageException e) {
            e.printStackTrace();
        } catch (OperationFailureException e) {
            e.printStackTrace();
            ex = e;
        }
```

Region Is Not Configured

If the region is not configured, then exception gives RFID_READER_REGION_NOT_CONFIGURED result the caller then gets supported regions and chooses operation regulatory region from list. Set region with required configurations.

```
if (ex.getResults() == RFIDResults.RFID_READER_REGION_NOT_CONFIGURED) {
// Get and Set regulatory configuration settings
// RegulatoryConfig regulatoryConfig = rfidReader.Config.getRegulatoryConfig();
// RegionInfo regionInfo =
            rfidReader.ReaderCapabilities.SupportedRegions.getRegionInfo(1);
// regulatoryConfig.setRegion(regionInfo.getRegionCode());
// rfidReader.Config.setRegulatoryConfig(regulatoryConfig);
        }
```

Reader Requires Bluetooth (BT)Password (RFD8500 only)

If the BT connection password is configured for connection, a direct call to the connection results with an exception pointing that the BT password is required. Set the password in the RFIDReader instance and then call connect again.

```
if (ex.getResults() == RFIDResults.RFID_CONNECTION_PASSWORD_ERROR) {
    // get password showPasswordDialog();
    rfidReader.setPassword(password);
    rfidReader.connect();
}
```

Reader Is Running In Batch Mode (RFD8500 only)

If the reader is configured with auto or batch mode enabled, the reader continues performing the inventory operation even it is not connected to an Android device. If, at that time, there is a connection made to a specific RFID reader, then the connection API throws the following exception. Then the application should stop the operation as required by usage.

```
if (ex.getResults() == RFIDResults.RFID_BATCHMODE_IN_PROGRESS) {
     // handle batch mode related stuff
}
```

Note that as the batch operation runs on the reader, the reader related information cannot be autonomously retrieved by the SDK. It is the responsibility of the application to retrieve/update reader related information using the following API once the BATCH operation is stopped before any other use.

```
rfidReader.PostConnectReaderUpdate();
```

Disconnect

When the application is done with the connection and operations on the RFID reader, it calls the following API to close the connection, and to release and clean up the resources.

```
rfidReader.disconnect();
```

NOTE: If a reader disconnection occurs, the reader.isConnected() flag may return the value false. If the application calls reader.connect(), the application should call reader.disconnect() regardless of the flag status.

Reconnect (RFD8500 only)

In case of abrupt disconnection from the reader, due to BT connectivity challenged in out of range scenarios, the application can use the following API to reconnect with reader. The application can retry periodically calling the reconnect API for finite number of times.

```
rfidReader.reconnect();
```

NOTE: When a reader reconnection occurs, it is advisable to retrieve the configuration again and registration of events, etc. It is also possible that the reader may have gone into batch mode if the reader was configured for that mode.

Dispose

When the application main activity is destroyed, it is required to dispose the SDK instance so that it can cleanly exit (unregistration and unbind by SDK as required).

```
readers.Dispose();
```

Knowing the Reader Capabilities

The capabilities (or Read-Only properties) of the reader are known using the ReaderCapabilites class. The reader capabilities include the following:

General Capabilities

- Model Name
- Number of antennas supported.
- Tag Event Reporting Supported - Indicates the reader's ability to report tag visibility state changes (New Tag, Tag Invisible, or Tag Visibility Changed).
- RSSI Filter Supported - Indicates the reader's ability to report tags based on the signal strength of the back-scattered signal from the tag.
- NXP Commands Supported - Indicates whether the reader supports NXP commands such as Change EAS, set Quiet, Reset Quiet, Calibrate.
- Tag LocationingSupported - Indicates the reader's ability to locate a tag.
- DutyCycleValues - List of DutyCycle percentages supported by the reader.

Gen2 Capabilities

- Block Erase - supported
- Block Write - supported
- State Aware Singulation - supported
- Maximum Number of Operation in Access Sequence
- Maximum Pre-filters allowable per antenna
- RF Modes.

Regulatory Capabilities

- Country Code
- Communication Standard.

UHF Band Capabilities

- Transmit Power table
- Hopping enabled
- Frequency Hop table - If hopping is enabled, this table has the frequency information.
- Fixed Frequency table - If hopping is not enabled, this table contains the frequency list used by the reader. The one-based position of a frequency in this list is its channel index.

Reader Identification

Reader ID and Reader ID Type (the reader identification can be MAC or EPC).

```
System.out.println("\nReader ID: " + reader.ReaderCapabilities.ReaderID.getID());
System.out.println("\nModelName: " + reader.ReaderCapabilities.getModelName());
System.out.println("\nCommunication Standard: " +
        reader.ReaderCapabilities.getCommunicationStandard().toString());
System.out.println("\nCountry Code: " + reader.ReaderCapabilities.getCountryCode());
System.out.println("\nRSSI Filter: " + reader.ReaderCapabilities.isRSSIFilterSupported());
System.out.println("\nTag Event Reporting: " +
        reader.ReaderCapabilities.isTagEventReportingSupported());
System.out.println("\nTag
        Locating Reporting: " + reader.ReaderCapabilities.isTagLocationingSupported());
System.out.println("\nNXP Command Support: " +
        reader.ReaderCapabilities.isNXPCommandSupported());
System.out.println("\nBlockEraseSupport: " +
reader.ReaderCapabilities.isBlockEraseSupported());
System.out.println("\nBlockWriteSupport: " +
        reader.ReaderCapabilities.isBlockWriteSupported());
System.out.println("\nBlockPermalockSupport:
        " + reader.ReaderCapabilities.isBlockPermalockSupported());
        System.out.println("\nRecommisionSupport: " +
                reader.ReaderCapabilities.isRecommisionSupported());
System.out.println("\nWriteWMISupport:
        " + reader.ReaderCapabilities.isWriteUMISupported());
        System.out.println("\nRadioPowerControlSupport: " +
reader.ReaderCapabilities.isRadioPowerControlSupported();System.out.println("\nHoppingEn
abled: " + reader.ReaderCapabilities.isHoppingEnabled());
System.out.println("\nStateAwareSingulationCapable: " +
        reader.ReaderCapabilities.isTagInventoryStateAwareSingulationSupported());
System.out.println("\nUTCClockCapable: " +
reader.ReaderCapabilities.isUTCClockSupported();System.out.println("\nNumOperationsInAcc
essSequence: " +
        reader.ReaderCapabilities.getMaxNumOperationsInAccessSequence());
System.out.println("\nNumPreFilters: " + reader.ReaderCapabilities.getMaxNumPreFilters());


System.out.println("\nNumAntennaSupported: " +
        reader.ReaderCapabilities.getNumAntennaSupported())
```

Configuring the Reader

RF Mode

The reader has one or more sets of C1G2 RF Mode that the reader is capable of operating. The supported RF mode is retrieved from the RF Mode table using the ReaderCapabilities class.

The API getRFModeTableInfo in reader capabilities.RFModes gets the RF Mode table from the reader. The getLinkedProfiles function described below populates the RFModeTable into an ArrayList.

```
// The rfModeTable is populated by the getRFModeTableInfo function
public RFModeTable rfModeTable =
        reader.ReaderCapabilities.RFModes.getRFModeTableInfo(0);
// The linked profiles are added to an Arraylist
private void getLinkedProfiles(ArrayList<String> linkedProfiles){
    RFModeTableEntry rfModeTableEntry = null;
    for (int i = 0; i < rfModeTable.length(); i++){
        rfModeTableEntry = rfModeTable.getRFModeTableEntryInfo(i);
        linkedProfiles.add(rfModeTableEntry.getBdrValue() + " " +
                rfModeTableEntry.getModulation() + " " + rfModeTableEntry.getPieValue() +
" " +
                fModeTableEntry.getMaxTariValue() + " " +
                rfModeTableEntry.getMaxTariValue() + " " +
rfModeTableEntry.getStepTariValue());
    }
}
```

Antenna Specific Configuration

The config class contains the Antennas object. The individual antenna is accessed and configured using the index.

Antenna Configuration

The AntennaProperties is used to set the antenna configuration to individual antenna or all the antennas.

The antenna configuration (SetAntennaConfig function) is comprised of Antenna ID, Receive Sensitivity

Index, Transmit Power Index, and Transmit Frequency Index. These indexes refer to the Receive Sensitivity table, Transmit Power table, Frequency Hop table, or Fixed Frequency table, respectively. These tables are available in Reader capabilities. getAntennaConfig function gets the antenna configuration for the given antenna ID.

RF Configuration

The function SetAntennaRFConfig is added to configure antenna RF configuration to individual antenna. This function is similar to SetAntennaConfig but includes additional parameters specific pertaining to the antenna.

The configuration includes Receive Sensitivity Index, Transmit Power Index, Transmit Frequency Index, and RF Mode Table Index. These indexes are refers to the Receive Sensitivity table, Transmit Power table, Frequency Hop table, Fixed Frequency table, or RF Mode table, respectively. These tables are available in Reader capabilities. Also, includes tari, transmit port, receive port and Antenna Stop trigger condition. The stop condition can be 'n' number of attempts, duration based.

The function getAntennaRFConfig gets the antenna RF configuration for the given antenna ID.

```
Antennas.AntennaRfConfig antennaRfConfig = reader.Config.Antennas.getAntennaRfConfig(1);
antennaRfConfig.setrfModeTableIndex(0);
antennaRfConfig.setTari(0);
antennaRfConfig.setTransmitPowerIndex(270);
reader.Config.Antennas.setAntennaRfConfig(1,antennaRfConfig);
```

Singulation Control

The function getSingulationControl retrieves the current settings of the singulation control from the reader for the given Antenna ID.

To set the singulation control settings, the setSingulationControll method is used. The following settings can be configured:

- Session: Session number to use for inventory operation.
- Tag Population: An estimate of the tag population in view of the RF field of the antenna.
- Tag Transit Time: An estimate of the time a tag typically remains in the RF field.
- State Aware Singulation Action: The action includes the Inventory state and SL flag. The action can be used if only the reader supports this capability. The ReaderCapabilities class helps to determine whether state-aware singulation is supported or not.

```
// Get Singulation Control for the antenna 1 Antennas.SingulationControl
singulationControl;
            singulationControl = reader.Config.Antennas.getSingulationControl(1);
// Set Singulation Control for the antenna 1 Antennas.SingulationControl
singulationControl;
            singulationControl.setSession(SESSION.SESSION_S0);
            singulationControl.setTagPopulation((short) 30);
            singulationControl.Action.setSLFlag(SL_FLAG.SL_ALL);
singulationControl.Action.setInventoryState(INVENTORY_STATE.INVENTORY_STATE_A);
            reader.Config.Antennas.setSingulationControl(1, singulationControl);
```

Tag Report Configuration

The SDK provides an ability to configure a set of fields to be reported in a response to an operation by a specific active RFID reader.

Supported fields that might be reported include the following:

- First seen time
- Last seen time

- PC value
- RSSI value
- Phase value
- Channel index
- Tag seen count.

The function `getTagStorageSettings` retrieves the tag report parameters from the reader for the given Antenna ID.

To set the Tag report parameters, the `setTagStorageSettings` method is used. The following settings can be configured:

```
 // Get tag storage settings from the reader
           TagStorageSettings tagStorageSettings = reader.Config.getTagStorageSettings();
// set tag storage settings on the reader with all fields
            tagStorageSettings.setTagFields(TAG_FIELD.ALL_TAG_FIELDS);
            reader.Config.setTagStorageSettings(tagStorageSettings);
```

Regulatory Configuration

The SDK supports managing of regulatory related parameters of a specific active RFID reader.

Regulatory configuration includes the following:

- Code of selected region
- Hopping
- Set of enabled channels.

A set of enabled channels includes only such channels that are supported in the selected region. If hopping configuration is not allowed for the selected regions, a set of enabled channels is not specified.

Regulatory parameters could be retrieved and set via `getRegulatoryConfig` and `setRegulatoryConfig` API functions accordingly. The region information is retrieved using `getRegionInfo` API. The following example demonstrates retrieving of current regulatory settings and configuring the RFID reader to operate in one of supported regions.

```
// Get and Set regulatory configuration settings
RegulatoryConfig regulatoryConfig = reader.Config.getRegulatoryConfig();
RegionInfo regionInfo = reader.ReaderCapabilities.SupportedRegions.getRegionInfo(1);
regulatoryConfig.setRegion(regionInfo.getRegionCode());
regulatoryConfig.setIsHoppingOn(regionInfo.isHoppingConfigurable());
regulatoryConfig.setEnabledChannels(regionInfo.getSupportedChannels());
reader.Config.setRegulatoryConfig(regulatoryConfig);
```

Saving Configuration

Various parameters of a specific RFID reader configured via SDK are lost after the next power down. The SDK provides an ability to save a persistent configuration of RFID reader. The `saveConfig` API function can be used to make the current configuration persistent over power down and power up cycles. The following example demonstrates utilization of mentioned API functions.

```
// Saving the configuration
reader.Config.saveConfig();
```

Reset Configuration to Factory Defaults

The SDK provides a way to reset the RFID reader to the factory default settings. The `resetFactoryDefaults` API can be used to attain this functionality. Once this method is called, all the reader settings like events, singulation control, etc will be lost and the RFID reader reboots. A connected application shall lose connectivity to the reader and must connect back again and is required to redo the basic steps for initializing the reader. The following example demonstrates utilization of mentioned API function.

```
// Resetting the configuration
reader.Config.resetFactoryDefaults();
```

Managing Events

The Application can register for one or more events so as to be notified of the same when it occurs. There are two main events.

- `eventReadNotify` - Notifies whenever read tag event occurs with read tag data as argument. By default, the event comes with tag data. If not required, disable using function `setAttachTagDataWithReadEvent`.
- `eventStatusNotify` - Notifies whenever status event occurs with status event data as argument.

Table 1 Events

Android Events	Description
GPI_EVENT	Not supported in Android RFID SDK.
BUFFER_FULL_WARNING_EVENT	When the internal buffers are 90% full, this event is signaled
ANTENNA_EVENT	Not supported in Android RFID SDK.
INVENTORY_START_EVENT	Inventory Operation started. In case of periodic trigger, this event is triggered for each period.
INVENTORY_STOP_EVENT	Inventory Operation has stopped. In case of periodic trigger this event is triggered for each period.
ACCESS_START_EVENT	Not supported in Android RFID SDK.
ACCESS_STOP_EVENT	Not supported in Android RFID SDK.
DISCONNECTION_EVENT	Event notifying disconnection from the Reader. The Application can call reconnect method periodically to attempt reconnection or call disconnect method to cleanup and exit.
BUFFER_FULL_EVENT	When the internal buffers are 100% full, this event is signaled and tags are discarded in FIFO manner.
NXP_EAS_ALARM_EVENT	Not Supported in Android RFID SDK.
READER_EXCEPTION_EVENT	Event notifying that an exception has occurred in the Reader. When this event is signaled, StatusEventData. ReaderExceptionEventData. ReaderExceptionEventType can be called to know the reason for the exception which is coming as part of Events.StatusEventArgs. The Application can continue to use the connection if the reader renders is usable.

Table 1 Events

Android Events	Description
HANDHELD_TRIGGER_EVENT	A hand-held Gun/Button event Pull/Release has occurred.
TEMPERATURE_ALARM_EVENT	When Temperature reaches Threshold level, this event is generated. The event data contains source name (PA/Ambient),current Temperature and alarm Level (Low, High or Critical)
BATTERY_EVENT	Events notifying different levels of battery, state of the battery, if charging or discharging.
OPERATION_END_SUMMARY_EVENT	Event generated when operation end summary has been generated. The data associated with the event contains total rounds, total number of tags and total time in micro secs.
BATCH_MODE_EVENT (RFD8500 only)	The batch mode event notification when the reader indicates whether batch mode is in progress.
POWER_EVENT	Events which notify the different power states of the reader device. The event data contains cause, voltage, current and power.

`registering for read tag data notification`

```
EventHandler eventHandler = new EventHandler();
            reader.Events.addEventsListener(eventHandler);
// Subscribe required status notification
            reader.Events.setInventoryStartEvent(true);
            myReader.Events.setInventoryStopEvent(true);
// enables tag read notification. if this is set to false, no tag read notification is send
            myReader.Events.setTagReadEvent(true);
            myReader.Events.setReaderDisconnectEvent(true);
            myReader.Events.setBatteryEvent(true);
```

Read/Status Notify handler

Implement the RfidEventsLister class to receive event notifications

```
class EventHandler implements RfidEventsListener {
    // Read Event Notification
    public void eventReadNotify(RfidReadEvents e){
// Recommended to use new method getReadTagsEx for better performance in case of large
       // tag population
        TagData[] myTags = myReader.Actions.getReadTags(100);
        if (myTags != null) {
            for (int index = 0; index < myTags.length; index++){
                System.out.println("Tag ID " + myTags[index].getTagID());
                if (myTags[index].getOpCode() ==
ACCESS_OPERATION_CODE.ACCESS_OPERATION_READ
                        &&
                        myTags[index].getOpStatus() ==
                                ACCESS_OPERATION_STATUS.ACCESS_SUCCESS) {
                    if (myTags[index].getMemoryBankData().length() > 0) {
                        System.out.println(" Mem Bank Data " +
                                myTags[index].getMemoryBankData());
                    }
                }
            }
        }
    }
    // Status Event Notification
    public void eventStatusNotify(RfidStatusEvents e) {
        System.out.println("Status Notification: " +
                e.StatusEventData.getStatusEventType());
    }
}
```

Unregistering for read tag data notification

```
reader.Events.removeEventsListener(eventHandler);
```

Device Status Related Events

Device status, such as battery, power, and temperature, is obtained through events after initiating the following API:

```
reader.Config.getDeviceStatus(battery, power, temperature)
```

Response to the above API comes as battery event, power event, and temperature event according to the set boolean value in the respective parameters.

The following is an example of how to get these events.

```
try {
    if (reader != null)
        reader.Config.getDeviceStatus(true, false, false);
    else
        stopTimer();
} catch (InvalidUsageException e) {
    e.printStackTrace();
} catch (OperationFailureException e) {
    e.printStackTrace();
}
```

Basic Operations

Tag Storage Settings

This section covers the basic operations that an application needs to perform on an RFID reader which includes inventory and single tag access operations.

The application needs to get the tags from the dll which are reported by the reader. Tags can be reported as part of an Inventory operation (reader.Actions.Inventory.perform) or a Read Access operation (reader.Actions.TagAccess.readEvent or reader.Actions.TagAccess.readWait).

Applications can also configure to receive tag reports that indicate the results of access operations as shown below.

```
TagStorageSettings tagStorageSettings = reader.Config.getTagStorageSettings();
reader.Config.setTagStorageSettings(tagStorageSettings);
```

Each tag has a set of associated information along with it. During the Inventory operation, the reader reports theEPC-ID of the tag, where as during the Read-Access operation the requested Memory Bank Data is also reportedapart from EPC-ID. In either case, there is additional information such as PC-bits, RSSI, last time seen, tag seen count, etc. that is available for each tag. This information is reported to the application as TagData for each tagreported by the reader.

Applications can also choose to enable/disable reporting certain fields in TAG_DATA. Disabling certain fields cansometimes improve the performance as the reader and the sdk are not processing that information. It can also resultin specific behavior. For example, disabling reporting an Antenna Id can result in the application receiving a single unique tag even though they were multiple entries of the same tag reported from different antennas. The

following demonstrates enabling the reporting of PeakRSSI, Tag Seen Count, PC and CRC only and disabling other fields such as Antenna ID, Time Stamps, and XPC.

```
TagStorageSettings tagStorageSettings = reader.Config.getTagStorageSettings();
TAG_FIELD[] tagField = new TAG_FIELD[4];
tagField[0] = TAG_FIELD.PC; tagField[1] =
        TAG_FIELD.PEAK_RSSI; tagField[2] =
        TAG_FIELD.TAG_SEEN_COUNT; tagField[3] =
        TAG_FIELD.CRC;
tagStorageSettings.setTagFields(tagField);
reader.Config.setTagStorageSettings(tagStorageSettings);
```

Tag Storage Use Cases

Example use-cases that get tags from the reader are as follows:

Simple Inventory (Continuous)

A Simple Continuous Inventory operation reads all tags in the field of view of all antennas of the connected RFID reader. It uses no filters (pre-filters or post-filters) and the start and stop trigger for the inventory is the default (for example, start immediately when reader.Actions.Inventory.perform is called, and stop immediately when reader.Actions.Inventory.stop is called).

```
// perform simple inventory
            reader.Actions.Inventory.perform();
// Keep getting tags in the eventReadNotify event if registered
// stop the inventory
            reader.Actions.Inventory.stop();
```

Simple Access Operations - On Single Tag

Tag Access operations are performed on a specific tag or applied on tags that match a specific Access-Filter. If no Access-Filter is specified, the Access Operation is performed on all tags in the field of view of chosen antennas.

This section covers the Simple Tag Access operation on a specific tag which is in the field of view of any of the antennas of the connected RFID reader.

`dpo should be disabled before any access operation for the RFD8500 reader.`

Read

The application calls method reader.Actions.TagAccess.readWait to read data from a specific memory bank.

```
// Read user memory bank for the given tag ID

String tagId = "1234ABCD00000000000025B1";
TagAccess tagAccess = new TagAccess();
TagAccess.ReadAccessParams readAccessParams = tagAccess.new ReadAccessParams();
TagData readAccessTag;
readAccessParams.setAccessPassword(0);
readAccessParams.setCount(4); // read 4 words
readAccessParams.setMemoryBank(MEMORY_BANK.MEMORY_BANK_USER);
readAccessParams.setOffset(0); // start reading from word offset 0
readAccessTag = reader.Actions.TagAccess.readWait(tagId, readAccessParams, null);
System.out.println(readAccessTag.getMemoryBank().toString() + " : " +
        readAccessTag.getMemoryBankData());
```

Write

The application calls method reader.Actions.TagAccess.writeWait or reader. Actions.
TagAccess. blockWriteWait to write data to a specific memory bank. The response is returned as a Tagdata from where a number of words can be retrieved.

```
                              // Write user memory bank data
TagData tagData = null;
      String tagId = "1234ABCD000000000000025B1";
      TagAccess tagAccess = new TagAccess();
      TagAccess.WriteAccessParams writeAccessParams = tagAccess.new
      WriteAccessParams();
      String writeData = "11223344"; // write data in string
      writeAccessParams.setAccessPassword(0);
      writeAccessParams.setMemoryBank(MEMORY_BANK.MEMORY_BANK_USER);
      writeAccessParams.setOffset(0); // start writing from word offset 0
      writeAccessParams.setWriteData(writeData);
      // antenna Info is null – performs on all antenna
      reader.Actions.TagAccess.writeWait(tagId, writeAccessParams, null, tagData);
```

The following shows usage of block write:

NOTE: The same write access parameters are passed as used above to easily switch between two APIs.

```
reader.Actions.TagAccess.blockWriteWait(tag Id, writeAccessParams, null, tagData);
```

Block-Write

The following shows usage of block write:

NOTE: The same write access parameters are passed as used above to easily switch between two APIs.

```
reader.Actions.TagAccess.blockWriteWait(tag Id, writeAccessParams, null, tagData);
```

Lock

The application calls method reader.Actions.TagAccess.lockWait to perform a lock operation on one or more memory banks with specific privileges.

```
// Lock the tag
String tagId = "1234ABCD000000000000025B1";
      TagAccess tagAccess = new TagAccess();
      TagAccess.LockAccessParams lockAccessParams = tagAccess.new
LockAccessParams();
/* lock now */
      lockAccessParams.setLockPrivilege(LOCK_DATA_FIELD.LOCK_USER_MEMORY,
         LOCK_PRIVILEGE.LOCK_PRIVILEGE_READ_WRITE);
      lockAccessParams.setAccessPassword(0);
      reader.Actions.TagAccess.lockWait(tagId, lockAccessParams, null);
```

Kill

The application calls method reader.Actions.TagAccess.killWait to kill a tag.

```
// Kill the tag
String tagId = "1234ABCD00000000000025B1";
TagAccess tagAccess = new TagAccess();
TagAccess.KillAccessParams killAccessParams = tagAccess.new KillAccessParams();
killAccessParams.setKillPassword(0);
reader.Actions.TagAccess.killWait(tagId, killAccessParams,null);
```

Block Erase

The application calls RFID_BlockErase to erase the contents of a tag.

```
// Block Erase
TagData tagData = new TagData ();
String tagId = "1234ABCD00000000000025B1";
TagAccess.BlockEraseAccessParams blockEraseAccessParams = tagAccess.new
    BlockEraseAccessParams();
blockEraseAccessParams.setAccessPassword(0);
blockEraseAccessParams.setMemoryBank(MEMORY_BANK.MEMORY_BANK_USER); // user memory bank
blockEraseAccessParams.setOffset(0); // start erasing from word offset 0
blockEraseAccessParams.setCount(8); // number of words to erase
mConnectedReader.Actions.TagAccess.blockEraseWait(tagId,
    blockEraseAccessParams, null,tagData);
```

Block-Permalock

The application calls method Reader.Actions.TagAccess.blockPermalockWait to block a permalock tag. Tags reported as part of Block-Permalock access operation have TagData.getOpCode as ACCESS_OPERATION_BLOCK_PERMALOCK and TagData.getOpStatus indicating the result of the operation; if TagData.OpStatus is ACCESS_SUCCESS, TagData.getMemoryBankData contains the Block-Permalock Mask Data.

```
// Block-Perma Lock the tag
String tagId = "1234ABCD00000000000025B1";
TagAccess tagAccess = new TagAccess();
TagAccess.BlockPermalockAccessParams blockPermalockAccessParams = tagAccess.new
    BlockPermalockAccessParams();
byte[] permalockMask = new byte[] {(byte)0xF0, 0x00};
blockPermalockAccessParams.setReadLock(true);
blockPermalockAccessParams.setMemoryBank(MEMORY_BANK.MEMORY_BANK_USER);
blockPermalockAccessParams.setOffset(0); // start BlockPermalock from word offset 0
blockPermalockAccessParams.setCount(1); // start BlockPermalock from word offset 0
blockPermalockAccessParams.setMask(permalockMask);
blockPermalockAccessParams.setMaskLength(2);
reader.Actions.TagAccess.blockPermalockWait(tagId, blockPermalockAccessParams, null);
```

Synchronous Access Operation API Updates

All synchronous access operation APIs have been overloaded with new parameters for prefilter support.

APIs: readWait, writeWait, killWait, lockWait, blockWriteWait, blockEraseWait and blockPermalockWait

```
public TagData readWait(java.lang.String tagID,
                        TagAccess.ReadAccessParams readAccessParams,
                        AntennaInfo antennaInfo, boolean bPrefilter)
```

Example:

```
reader.Actions.TagAccess.readWait("2F2203447334C3100002EA55", readAccessParams,
null,true);
```

writeWait and blockWriteWait APIs have additional parameter for specifying whether to use TID memory bank as prefilter.

```
writeWait(java.lang.String tagID,
          TagAccess.WriteAccessParams writeAccessParams,
          AntennaInfo antennaInfo,
          TagData tagData,
          boolean bPrefilter, boolean bTIDPrefilter)
```

Example:

Apply EPC memory bank as prefilter:

```
reader.Actions.TagAccess.writeWait(tagId, writeAccessParams, null, tagData, true, false);
```

Apply TID memory bank as prefilter:

```
reader.Actions.TagAccess.writeWait(tagId, writeAccessParams, null, tagData, true, true);
```

Provision of TID memory bank as prefilter and giving number of retries will help doing retries on partial writes of EPC memory bank.
Note - Using TID memory bank as prefilter will result in longer time of write, as SDK internally retrieves TID memory bank of tag.
Note - When prefilter is enabled, SDK does access operation in session S0 and uses `INV_B_NOT_INV_A_OR_DSRT_SL_NOT_ASRT_SL with INV_B inventory state`

`TagAccess.WriteAccessParams` have additional parameter to define number of retries.

```
void setWriteRetries(int writeRetries)
```

Number of retries to be performed for write access operation.
In case of failure SDK retries to perform write operation from data offset from last successful number of words written.
API to configure access operation time out via Config class.

Earlier version of SDK has hardcoded timeout of five seconds which results in longer wait for operation result to come if tag is not found in FOV or other reasons (not singulated).

By this application can shorten wait time and quickly retry in failure cases.

```
void setAccessOperationWaitTimeout(int timeout)
```

Method to set access operation timeout for all synchronous tag access operations APIs.

Example:

```
reader.Config.setAccessOperationWaitTimeout(1000);
```

Advanced Operations

Using Pre-Filters

Pre-filters are the same as the Select command of C1G2 specification. Once applied, pre-filters are applied prior to Inventory and Access operations.

Singulation

Singulation refers to the method of identifying an individual Tag in a multiple-Tag environment. RFID readers support State-Aware or State-Unaware pre-filtering (or singulation) which is indicated by the boolean flag IsTagInventoryStateAwareSingulationSupported in the ReaderCapabilities class.

In order to filter tags that match a specific condition, it is necessary to use the tag-sessions and their states (setting the tags to different states based on match criteria - `reader.Actions.PreFilters.add`) so that while performing inventory, tags can be instructed to participate (`singulation -reader.Config.Antennas.setSingulationControl/ Reader.Config.Antennas.SingulationControl`) or not participate in the inventory based on their states.

Sessions and Inventoried Flags

Tags provide four sessions (denoted S0, S1, S2, and S3) and maintain an independent inventoried flag for each session. Each of the four inventoried flags has two values, denoted A and B. These inventoried flag of each session can be set to A or B based on match criteria using method `Reader.Actions.PreFilters.Add`.

Selected Flag

Tags provide a selected flag, SL, which is asserted or deasserted based on match criteria using method `Reader.Actions.PreFilters.Add`.

State-Aware Singulation

In state-aware singulation, the application specifies detailed controls for singulation: Action and Target. Action indicates whether matching Tags assert or de-assert SL (Selected Flag), or set their inventoried flag to A or to B. Tags conforming to the match criteria specified using the method `reader.Actions.PreFilters.add/` are considered matching and the remaining are non-matching. Target indicates whether to modify a tag's SL flag or its inventoried flag, and in the case of inventoried, it further specifies one of four sessions.

Applying Pre-Filters

The following are the steps to use pre-filters:

1. Add pre-filters.
2. Set appropriate singulation controls.
3. Perform Inventory or Access operation.

Add Pre-filters

Each RFID reader supports a maximum number of pre-filters per antenna as indicated by `ReaderCapabilities.getMaxNumPreFilters` which is known using the ReaderCapabilities. The application sets pre-filters using reader.Actions.PreFilters.add and removes using `reader.Actions.PreFilters.delete`.

State-Aware Settings

```
PreFilters filters = new PreFilters();
            PreFilters.PreFilter filter = filters.new PreFilter();
            byte[] tagMask = new byte[] { 0x12,0x11 };
            filter.setAntennaID((short)1);// Set this filter for Antenna ID 1
            filter.setTagPattern(tagMask);// Tags which starts with 0x1211
            filter.setTagPatternBitCount(tagMask.length * 8);
            filter.setBitOffset(32); // skip PC bits (always it should be in bit length)
            filter.setMemoryBank(MEMORY_BANK.MEMORY_BANK_EPC);
            filter.setFilterAction(FILTER_ACTION.FILTER_ACTION_STATE_AWARE); // use state
aware singulation/ Add state aware pre-filter
            filter.StateAwareAction.setTarget(TARGET.TARGET_INVENTORIED_STATE_S1); //
inventoried flag of session
           // S1 of matching tags to B

filter.StateAwareAction.setStateAwareAction(STATE_AWARE_ACTION.STATE_AWARE_ACTION_INV_B;
// not to select tags that match the criteria
            reader.Actions.PreFilters.add(filter);
// It is also required to set appropriate singulation control not to
// get tags with inventoried flag B for session 1
```

Set Appropriate Singulation Controls

Now that the pre-filters are set (for example, tags are classified into matching or non-matching criteria), the application needs to specify which tags participate in the inventory using `reader.Config.Antennas.setSingulationControl()` Singulation Control must be specified with respect to each antenna such as pre-filters.

State-Aware Singulation

```
// Set the singulation control
Antennas.SingulationControl s1_singulationControl =
        reader.Config.Antennas.getSingulationControl(1);
s1_singulationControl.setSession(SESSION.SESSION_S1);
s1_singulationControl.Action.setInventoryState(INVENTORY_STATE.INVENTORY_STATE_B);
s1_singulationControl.Action.setSLFlag(SL_FLAG.SL_FLAG_DEASSERTED);
s1_singulationControl.Action.setPerformStateAwareSingulationAction(true);
reader.Config.Antennas.setSingulationControl(1, s1_singulationControl);
```

Perform Inventory or Access operation

Inventory or Access operation when performed after setting pre-filters, use the tags filtered out of pre-filters for their operation.

Using Triggers

Triggers are the conditions that are satisfied in order to start or stop an operation (Inventory or Access Sequence). This information can be specified using TriggerInfo class. The application can also configure the Tag-Report trigger which indicates when to receive ‘n’ unique Tag-Reports from the Reader.

We have to use `Config.setStartTrigger` and `Config.setStopTrigge` APIs to set triggers on the reader.

The following are some use-cases of using TRIGGER_INFO:

- Periodic Inventory: Start inventory at a specified time for a specified duration repeatedly.

```
TriggerInfo triggerInfo = new TriggerInfo();
// start inventory at every 2 seconds
triggerInfo.StartTrigger.setTriggerType(START_TRIGGER_TYPE.START_TRIGGER_TYPE_PERIODIC);
            triggerInfo.StartTrigger.Periodic.setPeriod(2000); // perform inventory for 2
seconds
// stop trigger
triggerInfo.StopTrigger.setTriggerType(STOP_TRIGGER_TYPE.STOP_TRIGGER_TYPE_DURATION);
            triggerInfo.StopTrigger.setDurationMilliSeconds(200); // stop after 200
milliseconds
```

- Perform ‘n’ Rounds of Inventory with a timeout: Start condition can be any; Stop condition is to perform ‘n’ rounds of inventory and then stop or stop inventory after the specified timeout.

```
TriggerInfo triggerInfo = new TriggerInfo();
// start inventory immediate

triggerInfo.StartTrigger.setTriggerType(START_TRIGGER_TYPE.START_TRIGGER_TYPE_IMMEDIATE);
// stop trigger

triggerInfo.StopTrigger.setTriggerType(STOP_TRIGGER_TYPE.STOP_TRIGGER_TYPE_N_ATTEMPTS_WIT
H_TIMEOUT);
            triggerInfo.StopTrigger.NumAttempts.setN((short)3); // perform 3 rounds of
inventory
            triggerInfo.StopTrigger.NumAttempts.setTimeout(3000); // timeout after 3
seconds //
```

- Read 'n' tags with a timeout: Start condition could be any; Stop condition is to stop after reading 'n' tags or stop inventory after the specified timeout.

```
TriggerInfo triggerInfo = new TriggerInfo();
// start inventory immediate

triggerInfo.StartTrigger.setTriggerType(START_TRIGGER_TYPE.START_TRIGGER_TYPE_IMMEDIATE);
// stop trigger

triggerInfo.StopTrigger.setTriggerType(STOP_TRIGGER_TYPE.STOP_TRIGGER_TYPE_TAG_OBSERVATIO
N_WITH_TIMEOUT);
            triggerInfo.StopTrigger.TagObservation.setN((short)100); // stop inventory
after reading 100 tags
            triggerInfo.StopTrigger.TagObservation.setTimeout(3000); // timeout after 3
seconds
// report back all read tags after getting 100 unique tags or after 3 seconds
```

- Inventory based on hand-held trigger: Start inventory when the hand-held gun/button trigger is pulled, and stop inventory when the hand-held gun/button trigger is released or subject to timeout.

```
TriggerInfo triggerInfo = new TriggerInfo();

triggerInfo.StartTrigger.setTriggerType(START_TRIGGER_TYPE.START_TRIGGER_TYPE_HANDHELD);
// Start Inventory when the Handheld trigger is pressed

triggerInfo.StartTrigger.Handheld.setHandheldTriggerEvent(HANDHELD_TRIGGER_EVENT_TYPE.
                    HANDHELD_TRIGGER_PRESSED);

triggerInfo.StopTrigger.setTriggerType(STOP_TRIGGER_TYPE.STOP_TRIGGER_TYPE_HANDHELD_WITH_
TIMEOUT);
// Stop Inventory when the Handheld trigger is released

triggerInfo.StopTrigger.Handheld.setHandheldTriggerEvent(HANDHELD_TRIGGER_EVENT_TYPE.
                    HANDHELD_TRIGGER_RELEASED);
            triggerInfo.StopTrigger.Handheld.setHandheldTriggerTimeout(0);
```

- Set the trigger using the following APIs, perform inventory and other operations which are using above set start and stop triggers.

```
reader.Config.setStartTrigger(triggerInfo.StartTrigger);
reader.Config.setStopTrigger(triggerInfo.StopTrigger);
reader.Actions.Inventory.perform();
```

Access

Using Access-Filters

In order to perform an access operation on multiple tags, the application can set ACCESS_FILTER to filter the required tags. If ACCESS_FILTER is not specified, the operation is performed on all tags. In any case, the PRE_FILTER(s) (if any is set) applies prior to ACCESS_FILTER.

The following access-filter gets all tags that have zeroed reserved memory bank.

```
AccessFilter accessFilter = new AccessFilter();
              byte[] tagMask = new byte[] { (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff,
(byte)0xff,
                        (byte)0xff, (byte)0xff, (byte)0xff };
// Tag Pattern A
              accessFilter.TagPatternA.setMemoryBank(MEMORY_BANK.MEMORY_BANK_RESERVED);
              accessFilter.TagPatternA.setTagPattern(new byte[] { 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
                        0x00, 0x00 }); accessFilter.TagPatternA.setTagPatternBitCount(8 * 8);
              accessFilter.TagPatternA.setBitOffset(0);
              accessFilter.TagPatternA.setTagMask(tagMask);
              accessFilter.TagPatternA.setTagMaskBitCount(tagMask.length * 8);
              accessFilter.setAccessFilterMatchPattern(FILTER_MATCH_PATTERN.A);
```

Access Operation on Multiple Tags

Performing a single Access operation on multiple tags is an asynchronous operation. The function issues the access-operation and returns. The reader performs one round of inventory using pre-filters, if any, and then applies the access-filters and the resultant tags are subject to the access-operation. When the access operation is complete, the SDK signals the eventStatusNotify event with event data as INVENTORY_STOP_EVENT/.

In case of Read access operation (reader.Actions.TagAccess.readEvent/) the event eventReadNotify is signaled when tags are reported.

The following demonstrates a sample write-access operation: //
Create Event to signify access operation complete.

```
reader.Events.setInventoryStartEvent(true);
         reader.Events.setInventoryStopEvent(true); // Data Read Notification from the
reader
class EventHandler implements RfidEventsListener {
     // Read Event Notification
     public void eventReadNotify(RfidReadEvents e){
         TagData tag = e.getReadEventData().tagData;
         System.out.println("Tag ID " + tag.getTagID());
         if (tag.getOpCode() == ACCESS_OPERATION_CODE.ACCESS_OPERATION_READ &&
                 tag.getOpStatus() == ACCESS_OPERATION_STATUS.ACCESS_SUCCESS) {
             if (tag.getMemoryBankData().length() > 0) {
                 System.out.println(" Mem Bank Data " + tag.getMemoryBankData());
             }
         }
     }
     // Status Event Notification
     public void eventStatusNotify(RfidStatusEvents e) {
         if (e.StatusEventData.getStatusEventType() ==
                 STATUS_EVENT_TYPE.INVENTORY_START_EVENT) {
// Access operation started
         } else if(e.StatusEventData.getStatusEventType() ==
                 STATUS_EVENT_TYPE.INVENTORY_STOP_EVENT) {
// Access operation stopped - Can be used to signal waiting thread
         }
     }
}
```

```
// Access Filter - EPC ID starting with 0x1122
AccessFilter accessFilter = new AccessFilter();
            byte[] tagMask = new byte[] { 0xff, 0xff };
// Tag Pattern A
            accessFilter.TagPatternA.setMemoryBank(MEMORY_BANK.MEMORY_BANK_EPC);
            accessFilter.TagPatternA.setTagPattern(new byte[] { 0x11, 0x22});
            accessFilter.TagPatternA.setTagPatternBitCount(2 * 8);
            accessFilter.TagPatternA.setBitOffset(0);
            accessFilter.TagPatternA.setTagMask(tagMask);
            accessFilter.TagPatternA.setTagMaskBitCount(tagMask.length * 8);
            accessFilter.setAccessFilterMatchPattern(FILTER_MATCH_PATTERN.A);

// Write user memory bank data
            TagAccess tagAccess = new TagAccess();
            TagAccess.WriteAccessParams writeAccessParams = tagAccess.new
WriteAccessParams();
            String writeData = "ABCDABCD"
            writeAccessParams.setAccessPassword(0);
            writeAccessParams.setMemoryBank(MEMORY_BANK.MEMORY_BANK_USER);
            writeAccessParams.setOffset(0); // start writing from word offset 0
            writeAccessParams.setWriteData(writeData);
// Asynchronous write operation
            reader.Actions.TagAccess.writeEvent(writeAccessParams, accessFilter, null);
// wait for access operation to complete (INVENTORY_STOP_EVENT is signaled after
completing
            //the access operation in the eventStatusNotify)
```

Using Access Sequence

The application issues multiple access operations on a single go using Access-Sequence API. This is useful when each tag from a set of (access-filtered) tags is to be subject to an order of access operations.

The maximum number of access-operations that can be specified in an access sequence is available in `reader.ReaderCapabilities.getMaxNumOperationsInAccessSequence` of ReaderCapabilities class.

The operations are performed in the same order in which it is added to it sequence. An operation can be removed from the sequence using `reader.Actions.TagAccess.OperationSequence.delete` and finally de-initialized if no longer needed by calling the function.

```
reader.Actions.TagAccess.OperationSequence.deleteAll();
// add Write Access operation - Write to User memory
TagAccess tagAccess = new TagAccess();
TagAccess.Sequence opSequence = tagAccess.new Sequence(tagAccess);
TagAccess.Sequence.Operation op1 = opSequence.new Operation();
op1.setAccessOperationCode(ACCESS_OPERATION_CODE.ACCESS_OPERATION_WRITE);
op1.WriteAccessParams.setMemoryBank(MEMORY_BANK.MEMORY_BANK_USER);
op1.WriteAccessParams.setAccessPassword(0);
op1.WriteAccessParams.setOffset(0);
op1.WriteAccessParams.setWriteData("55667788");
op1.WriteAccessParams.setWriteDataLength(4);
reader.Actions.TagAccess.OperationSequence.add(op1);
// add Write Access operation - Write to Reserved memory bank
TagAccess.Sequence.Operation op2 = opSequence.new Operation();
op2.setAccessOperationCode(ACCESS_OPERATION_CODE.ACCESS_OPERATION_WRITE);
op2.WriteAccessParams.setMemoryBank(MEMORY_BANK.MEMORY_BANK_USER);
op2.WriteAccessParams.setAccessPassword(0);
op2.WriteAccessParams.setOffset(0);
op2.WriteAccessParams.setWriteData("BBBBCCCC");
op2.WriteAccessParams.setWriteDataLength(4);
reader.Actions.TagAccess.OperationSequence.add(op2);
// perform access sequence
reader.Actions.TagAccess.OperationSequence.performSequence();
// if the access operation is to be terminated without meeting stop trigger (ifspecified),
// stopSequence method can be called
reader.Actions.TagAccess.OperationSequence.stopSequence();
```

Gen2v2 Operations

This section covers the Gen2V2 operations that an application needs to performed on a RFID Reader which supports Gen2v2 commands such as authenticate, untraceable, and readbuffer.

Authenticate

Authenticate operation takes in the message data and message length with few of the options such as decision on including the response length, sending the response, etc.

The AuthenticateParams contain the message data, message length and other settings to be sent to the reader. The accessfilter parameter contains the tag pattern on which the operation occurs.

```
// authenticate
// Tag Pattern A
accessFilter.TagPatternA.setMemoryBank(MEMORY_BANK.MEMORY_BANK_EPC);
accessFilter.TagPatternA.setTagPattern(new byte[]{(byte)0xe2, (byte)0xc0 });
accessFilter.TagPatternA.setTagPatternBitCount(16);
accessFilter.TagPatternA.setBitOffset(32);
accessFilter.TagPatternA.setTagMask(tagMask);
accessFilter.TagPatternA.setTagMaskBitCount(tagMask.length*8);
accessFilter.setAccessFilterMatchPattern(FILTER_MATCH_PATTERN.A);
```

```
// G2V2 authenticate
// Gen2V2 gen2V2 - new Gen2v2 ();
Gen2v2.AuthenticateParams AuthenticateParams = gen2V2.new AuthenticateParams();
AuthenticateParams.setMsgData("2001FD5D8048F48DD09AAD22000111");
AuthenticateParams.setMsgLen(120);
AuthenticateParams.setIncrespLen(true);
AuthenticateParams.setStoreResp(false);
AuthenticateParams.setSentResp(true);
try {
    reader.Actions.gen2v2Access.authenticate(AuthenticateParams, accessFilter, null);
} catch (InvalidUsageException e) {
    e.printStackTrace();
} catch (OperationFailureException e) {
    e.printStackTrace();
}
```

Keep getting response in the eventReadNotify event if registered

The response and result in the Tagdata will contain the information obtained from the operation.

```
public class EventHandler implements RfidEventsListener {
    // Read Event Notification
    public void eventReadNotify(RfidReadEvents e) {
        TagData[] myTags = reader.Actions.getReadTags(100);
        if (myTags != null) {
            for (int index = 0; index < myTags.length; index++) {
                System.out.println("Tag ID " + myTags[index].getTagID());
                if ((myTags[index].getG2v2OpStatus()!= null) &&
                        (myTags[index].getG2v2OpStatus() ==
                                GEN2V2_OPERATION_STATUS.ACCESS_SUCCESS)) {
                    if(!myTags[index].getG2v2Response().isEmpty()){
                        System.out.println("Gen2v2 authenticate response " +
                                myTags[index].getG2v2Response());
                    }
                }
            }
        }
    }// Status Event Notification
    public void eventStatusNotify(RfidStatusEvents e) {
        System.out.println("Status Notification: " +
                e.StatusEventData.getStatusEventType());
        if (e.StatusEventData.getStatusEventType() ==
STATUS_EVENT_TYPE.INVENTORY_START_EVENT) {
// Access operation started
        } else if(e.StatusEventData.getStatusEventType() ==
                STATUS_EVENT_TYPE.INVENTORY_STOP_EVENT) {
// Access operation stopped - Can be used to signal waiting thread
        }
    }
}
```

Untraceable

Untraceable operation lets the user decide which memory bank to show and what length of the memory bank to show. Here the UntraceableParams contain the settings and password. The accessfilter parameter contains the tag pattern on which the operation occurs.

```
        // untraceable
                    AccessFilter accessFilter = new AccessFilter();
                    byte[] tagMask = new byte[]{(byte) 0xff, (byte) 0xff, };
        // Tag Pattern A
                    accessFilter.TagPatternA.setMemoryBank(MEMORY_BANK.MEMORY_BANK_EPC);
                    accessFilter.TagPatternA.setTagPattern("2f22");
                    accessFilter.TagPatternA.setTagPatternBitCount(32);
                    accessFilter.TagPatternA.setBitOffset(32);
                    accessFilter.TagPatternA.setTagMask(tagMask);
                    accessFilter.TagPatternA.setTagMaskBitCount(tagMask.length*8);
                    accessFilter.setAccessFilterMatchPattern(FILTER_MATCH_PATTERN.A);
                    Gen2v2 gen2V2 = new Gen2v2();
                    Gen2v2.UntraceableParams UntraceableParams = gen2V2.new
        UntraceableParams();
                    UntraceableParams.setPassword(0);
                    UntraceableParams.setShowEpc(true);
                    UntraceableParams.setHideEpc(false);
                    UntraceableParams.setShowUser(false);
                    UntraceableParams.setEpcLen(6);
                    UntraceableParams.setTid(UNTRACEABLE_TID.HIDE_ALL_TID);
                    try{
                  reader.Actions.gen2v2Access.untraceable(UntraceableParams, accessFilter,
        null);
                } catch (InvalidUsageException e) {
                    e.printStackTrace();
                } catch (OperationFailureException e) {
                    e.printStackTrace(); }
```

After this, when inventory is run, the effects of the settings sent are seen in untraceable operation.

Resetting the Reader

The RFID SDK supports performing soft-reset of the reader. A connected application shall lose connectivity to the reader, must connect back again, and is required to redo the basic steps for initializing the reader. The following example demonstrates utilization of then API function.

```
// Resetting the reader
reader.Actions.reset();
```

Tag Locationing

Readers that support the Tag Locationing feature report the same in the field isTagLocationingSupported of ReaderCapabilities as true. This feature is supported only on hand-held readers and is useful to locate a specific tag in the field of view of the reader’s antenna. The default locationing algorithm supported on the reader can perform locationing only on a single antenna. reader.Actions.TagLocationing.Perform can be used to start locating a tag, and reader.Actions.TagLocationing.Stop to stop the locationing operation. The result of locationing of a tag is reported as LocationInfo in TagData and is present in TagData if tagData isContainsLocationInfo is true. tagData.LocationInfo.getRelativeDistance gives the relative distance of the tag from the reader antenna.

Performing Tag Locationing on a particular tag ID.

```
    reader.Actions.TagLocationing.Perform("E2002849491502421020B330",null);
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                reader.Actions.Inventory.stop();
    //The response of the tag locationing comes through eventReadNotify in the following
                Event Handler.
                public class EventHandler implements RfidEventsListener {
                    // Read Event Notification
                    public void eventReadNotify(RfidReadEvents e) {
                        TagData[] myTags = reader.Actions.getReadTags(100);
                        if (myTags != null) {
                            for (int index = 0; index < myTags.length; index++) {
                                System.out.println("Tag ID " + myTags[index].getTagID());
                                if (myTags[index].isContainsLocationInfo()) {
                                    int tag = index;
                                    System.out.println("Tag locationing distance " +
                                            myTags[tag].LocationInfo.getRelativeDistance());
                                }
                            }
                        }
                    }
                }
```

Trigger Mode - RFID and Barcode

To set the trigger mode to work as RFID or Barcode functionality, use the following API.

```
rfidReader.Config.setTriggerMode(ENUM_TRIGGER_MODE.RFID_MODE, true);
```

First parameter is mode enum value and second parameter indicates whether SDK should take care of disabling scanner plugin. When second parameter is true; SDK executes code to disable scanner plugin.

NOTE: It is recommended that Application handle scanner plugin enable/disable in pause and resume activity respectively to avoid cross triggering to functionality not required.

Refer to the DataWedge documentation at the following locations:

techdocs.zebra.com/datawedge/6-5/guide/api/scannerinputplugin/

techdocs.zebra.com/datawedge/6-5/guide/api/resultinfo/

The following code demonstrates ways to disable scanner plugin.

```
private void EnableDisableScannerPlugIn(boolean enable)
    {
// define action and data strings
        String scannerInputPlugin = "com.symbol.datawedge.api.ACTION";
        String extraData = "com.symbol.datawedge.api.SCANNER_INPUT_PLUGIN";
// following flag is updated in RESULT INFO received for scanner status
        mScannerStatusReceived = false;
        if (enable) {
// enable scanner plugin
// create the intent
            Intent i = new Intent();
// set the action to perform
            i.setAction(scannerInputPlugin);
// add additional info
            i.putExtra(extraData, "ENABLE_PLUGIN");
//
            i.putExtra("SEND_RESULT", "true");
            i.putExtra("COMMAND_IDENTIFIER", "RFIDDEMO_ENABLE_SCANNER");
// send the intent to DataWedge
            this.sendBroadcast(i);
        } else {
//disable scanner plugin
// create the intent
            Intent i = new Intent();
// set the action to perform
            i.setAction(scannerInputPlugin);
// add additional info
            i.putExtra(extraData, "DISABLE_PLUGIN");
   //
                i.putExtra("SEND_RESULT", "true");
                i.putExtra("COMMAND_IDENTIFIER", "RFIDEMO_DISABLE_SCANNER");
   // send the intent to DataWedge
                this.sendBroadcast(i);
            }
   // wait for synchronization response
            int timeout = 0;
            while (timeout++ < 10 && !mScannerStatusReceived) {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            Log.d(TAG, "setTriggerMode synchronization done");
        }
```

Set Attribute

The RFID SDK supports setting RFID parameters through attributes. Please refer to the respective reader model integrator guide for information about various types of supported attributes and details.

The following code demonstrates ways to set reader LED Mode indication.

```
SetAttribute setAttributeinfo = new SetAttribute();
setAttributeinfo.setAttvalue("2");
setAttributeinfo.setAtttype("B");
setAttributeinfo.setAttnum(1785);
rfidReader.Config.setAttribute(setAttributeinfo);
```

Set Host LED Support

The RFID SDK supports setting host LED indication for tag read events using following APIs:

```
reader.Config.setLedBlinkEnable(true);
```

Set Default Configuration

This API helps to rapidly set default configuration on the reader.

API supports the following reader configurations:

- Antenna configuration
- Singulation settings
- Tag storage settings
- Delete all pre-filters (select record) on device
- Dynamic power setting (DPO enable/disable)
- Any five attributes passed as array of attributes

- API sets start and stop trigger type to immediate by itself.

```
// antenna power
Antennas.AntennaRfConfig antennaRfConfig =
        mConnectedReader.Config.Antennas.getAntennaRfConfig(1);
//mConnectedReader.Config.Antennas.new AntennaRfConfig();
antennaRfConfig.setTransmitPowerIndex(270);

// singulation control

Antennas.SingulationControl singulationControl =
        mConnectedReader.Config.Antennas.getSingulationControl(1);
singulationControl.Action.setInventoryState(INVENTORY_STATE.INVENTORY_STATE_A);
singulationControl.Action.setPerformStateAwareSingulationAction(false);
// Tag storage settings

TagStorageSettings tagStorageSettings = new TagStorageSettings();
TAG_FIELD[] tagfields = new TAG_FIELD[2];
tagfields[0] = PEAK_RSSI;
tagfields[1] = TAG_SEEN_COUNT;
tagStorageSettings.setTagFields(tagfields);

// following is attribute to set trigger mode as RFID

SetAttribute[] setAttributeArray = new SetAttribute[1];
setAttributeArray[0] = new SetAttribute(1644,"B","0",0);

// API call
mConnectedReader.Config.setDefaultConfigurations(antennaRfConfig, singulationControl,
        tagStorageSettings, true, true, setAttributeArray);
```

Exceptions

The Zebra RFID Android SDK throws two types of exceptions as a given:

- InvalidUsageException: This exception is thrown when the user passes an invalid parameter, calling getInfo() gives detail error message.
- OperationFailureException: This exception is thrown when the requested operation is failed. The Exception contains the Operation RFIDResults, status description, time stamp & vendor specific message for the operation failure.

Exception Handling

All API should be called under try-catch block to catch the exception thrown while performing API by SDK.

```
try {
    reader.connect();
} catch (InvalidUsageException e) {
    e.printStackTrace();
} catch (OperationFailureException e) {
    e.printStackTrace();
}
```

General Guidelines

Synchronization

Use synchronization method to cover connection, configuration, and disconnection blocks.

Connection related APIs:

```
synchronized (lock) {
                readers.GetAvailableRFIDReaderList();
                reader.Connect();
// Code related to configuration and initial setup after connection
                ConfigureRfidReader();
            }
```

Disconnection related APIs:

```
synchronized (lock) {
    reader.disconnect();
    readers.Dispose();
}
```

Threading

The following APIs must be called from background thread:

- `GetAvailableRFIDReaderList`
- connect
- disconnect
- Dispose

Quick Start Sample

This code provides quick implementation of MainActivity with RFID SDK and inventory on hand-held trigger.

```
import android.os.AsyncTask;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.widget.TextView;
import android.widget.Toast;


import com.zebra.rfid.api3.ACCESS_OPERATION_CODE;
import com.zebra.rfid.api3.ACCESS_OPERATION_STATUS;
import com.zebra.rfid.api3.ENUM_TRANSPORT;
import com.zebra.rfid.api3.ENUM_TRIGGER_MODE;
import com.zebra.rfid.api3.HANDHELD_TRIGGER_EVENT_TYPE;
import com.zebra.rfid.api3.InvalidUsageException;
import com.zebra.rfid.api3.OperationFailureException;
import com.zebra.rfid.api3.RFIDReader;
import com.zebra.rfid.api3.ReaderDevice;
import com.zebra.rfid.api3.Readers;
import com.zebra.rfid.api3.RfidEventsListener;
import com.zebra.rfid.api3.RfidReadEvents;
import com.zebra.rfid.api3.RfidStatusEvents;
import com.zebra.rfid.api3.START_TRIGGER_TYPE;
import com.zebra.rfid.api3.STATUS_EVENT_TYPE;
import com.zebra.rfid.api3.STOP_TRIGGER_TYPE;
import com.zebra.rfid.api3.TagData;
import com.zebra.rfid.api3.TriggerInfo;

import java.util.ArrayList;



public class MainActivity extends AppCompatActivity {

        private static Readers readers;
        private static ArrayList<ReaderDevice> availableRFIDReaderList;
        private static ReaderDevice readerDevice;
        private static RFIDReader reader;
        private static String TAG = "DEMO";
        TextView textView;
        private EventHandler eventHandler;
    @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                // UI
                textView = (TextView) findViewById(R.id.TagText);
                // SDK
       if (readers == null) {
       readers = new Readers(this, ENUM_TRANSPORT.SERVICE_SERIAL);
       }
                new AsyncTask<Void, Void, Boolean>() {
```

```
                @Override
                protected Boolean doInBackground(Void... voids) {
                    try {
                        if (readers != null) {
                            if (readers.GetAvailableRFIDReaderList() != null) {
                                availableRFIDReaderList =
                                readers.GetAvailableRFIDReaderList();
                                if (availableRFIDReaderList.size() != 0) {
                                    // get first reader from list
                                    readerDevice = availableRFIDReaderList.get(0);
                                    reader = readerDevice.getRFIDReader();
                                    if (!reader.isConnected()) {
                                        // Establish connection to the RFID Reader
                                        reader.connect();
                                        ConfigureReader();
                                        return true;
                                    }
                                }
                            }
                        }
                    } catch (InvalidUsageException e) {
                        e.printStackTrace();
                    } catch (OperationFailureException e) {
                        e.printStackTrace();
                        Log.d(TAG, "OperationFailureException " + e.getVendorMessage());
                    }
                    return false;
                }
                @Override
                protected void onPostExecute(Boolean aBoolean) {
                    super.onPostExecute(aBoolean);
                    if (aBoolean) {
                        Toast.makeText(getApplicationContext(), "Reader Connected",
Toast.LENGTH_LONG).show();
                        //textView.setText("Reader connected");
                    }
                }
            }.execute();

        }
    private void ConfigureReader() {
            if (reader.isConnected()) {
                TriggerInfo triggerInfo = new TriggerInfo();

triggerInfo.StartTrigger.setTriggerType(START_TRIGGER_TYPE.START_TRIGGER_TYPE_IMMEDIATE);

triggerInfo.StopTrigger.setTriggerType(STOP_TRIGGER_TYPE.STOP_TRIGGER_TYPE_IMMEDIATE);
                try {
                    // receive events from reader
                    if (eventHandler == null)
                        eventHandler = new EventHandler();
                    reader.Events.addEventsListener(eventHandler);
                    // HH event
                    reader.Events.setHandheldEvent(true);
                    // tag event with tag data
```

```
reader.Events.setTagReadEvent(true);
                reader.Events.setAttachTagDataWithReadEvent(false);
                // set trigger mode as rfid so scanner beam will not come
                reader.Config.setTriggerMode(ENUM_TRIGGER_MODE.RFID_MODE, true);
                // set start and stop triggers
                reader.Config.setStartTrigger(triggerInfo.StartTrigger);
                reader.Config.setStopTrigger(triggerInfo.StopTrigger);
            } catch (InvalidUsageException e) {
                e.printStackTrace();
            } catch (OperationFailureException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        try {
            if (reader != null) {
                reader.Events.removeEventsListener(eventHandler);
                reader.disconnect();
                Toast.makeText(getApplicationContext(), "Disconnecting reader",
Toast.LENGTH_LONG).show();
                reader = null;
                readers.Dispose();
                readers = null;
            }
        } catch (InvalidUsageException e) {
            e.printStackTrace();
        } catch (OperationFailureException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }


// Read/Status Notify handler
    // Implement the RfidEventsLister class to receive event notifications
    public class EventHandler implements RfidEventsListener {
        // Read Event Notification
        public void eventReadNotify(RfidReadEvents e) {
           // Recommended to use new method getReadTagsEx for better performance in case of
large tag population
            TagData[] myTags = reader.Actions.getReadTags(100);
            if (myTags != null) {
                for (int index = 0; index < myTags.length; index++) {
                    Log.d(TAG, "Tag ID " + myTags[index].getTagID());
                    if (myTags[index].getOpCode() ==
ACCESS_OPERATION_CODE.ACCESS_OPERATION_READ &&
                            myTags[index].getOpStatus() ==
ACCESS_OPERATION_STATUS.ACCESS_SUCCESS) {
                        if (myTags[index].getMemoryBankData().length() > 0) {
                          Log.d(TAG, " Mem Bank Data " + myTags[index].getMemoryBankData());
```

```
                    }
                }
            }
        }
    }

    // Status Event Notification
    public void eventStatusNotify(RfidStatusEvents rfidStatusEvents) {
        Log.d(TAG, "Status Notification: " +
rfidStatusEvents.StatusEventData.getStatusEventType());
        if (rfidStatusEvents.StatusEventData.getStatusEventType() ==
STATUS_EVENT_TYPE.HANDHELD_TRIGGER_EVENT) {
            if
(rfidStatusEvents.StatusEventData.HandheldTriggerEventData.getHandheldEvent() ==
HANDHELD_TRIGGER_EVENT_TYPE.HANDHELD_TRIGGER_PRESSED) {
                new AsyncTask<Void, Void, Void>() {
                    @Override
                    protected Void doInBackground(Void... voids) {
                        try {
                            reader.Actions.Inventory.perform();
                        } catch (InvalidUsageException e) {
                            e.printStackTrace();
                        } catch (OperationFailureException e) {
                            e.printStackTrace();
                        }
                        return null;
                    }
                }.execute();
            }
if (rfidStatusEvents.StatusEventData.HandheldTriggerEventData.getHandheldEvent() ==
HANDHELD_TRIGGER_EVENT_TYPE.HANDHELD_TRIGGER_RELEASED) {
                new AsyncTask<Void, Void, Void>() {
                    @Override
                    protected Void doInBackground(Void... voids) {
                        try {
                            reader.Actions.Inventory.stop();
                        } catch (InvalidUsageException e) {
                            e.printStackTrace();
                        } catch (OperationFailureException e) {
                            e.printStackTrace();
                        }
                        return null;
                    }
                }.execute();
            }
        }
    }
  }
}
```

ZEBRA RFID SDK for Xamarin Android

Introduction

This chapter provides detailed information about how to use various basic and advanced functionality to develop an Xamarin Android application using the Zebra RFID SDK for Xamarin Android.

The Zebra RFID SDK for Xamarin allows applications to communicate with RFID readers connected to a mobile device.

The Zebra RFID SDK for Android provides the API that can be used by external applications to manage connection of the RFID readers, and to control connected RFID reader.

Basics

Zebra RFID Xamarin consists of library in dll format that should be linked with an external Xamarin application. All available APIs are defined under the Com.Zebra.Rfid.Api3 namespace. RFIDReader is the root class in the SDK. The application uses a single instance of an RFIDReader object to interact with a particular reader. Use available readers and the RFIDReader object to register for events, connect with readers, and after successful connection, perform required operations such as inventory, access, and locate. It is recommended that all API calls are made using Xamarin `ThreadPool.QueueUserWorkItem` so that operations are performed in the background thread keeping the UI thread free. If the API call fails, the API throws an exception. The application calls all APIs in try-catch block for handling exceptions.

Connection Management

Connect to an RFID Reader

Connection is the first step to talk to an RFID reader. Importing package is the first step to use any API. Import the package as follows:

```
using Com.Zebra.Rfid.Api3;
```

As first step create instance of Readers class with passing activity context as first parameter and enum value ServiceSerial as second parameter

```
readers = new Readers(this.context, ENUM_TRANSPORT.ServiceSerial);
```

The Readers class instance gives a list of all available/paired RFID readers with an Android device. Readers list is in the form of ReaderDevice class.

```
IList<ReaderDevice> readersList = readers.AvailableRFIDReaderList;
 ReaderDevice readerDevice = readersList[0];
```

ReaderDevice class includes instance of RFIDReader class; which is root class to interface and performing all operations with RFID reader.

```
rfidReader.Connect();
```

In addition, the application can implement `Readers.IRFIDReaderEventHandler` in the following way to get notified for RFID reader getting attached (added) / detached (removal).

```
public class MainActivity: Activity, Java.Lang.Object, Readers.IRFIDReaderEventHandler
{
       public void RFIDReaderAppeared(ReaderDevice readerDevice)
        {
            // handle reader addition
        }
        public void RFIDReaderDisappeared(ReaderDevice readerDevice)
        {
                        // handle reader removal
        }
}
```

Special Connection Handling Cases

In a normal scenario, the reader connects fine, but the following are the cases which require special handling at the time of connection.

The following example demonstrates the connection is handled under try-catch block and OperationFailure exception is thrown by connection API is stored and used for further analysis.

```
private OperationFailureException Ex;
        try {
              // Establish connection to the RFID Reader
            RfidReader.Connect();
        } catch (InvalidUsageException E) {
            E.PrintStackTrace();
        } catch (OperationFailureException E) {
            E.PrintStackTrace();
            Ex = E;
        }
```

Region Is Not Configured

If the region is not configured, then exception gives RfidReaderRegionNotConfigured result the caller then gets supported regions and chooses operation regulatory region from list. Set region with required configurations.

```
if (ex.Results == RFIDResults.RfidReaderRegionNotConfigured) {
// Get and Set regulatory configuration settings
// RegulatoryConfig regulatoryConfig = rfidReader.Config.RegulatoryConfig;
// RegionInfo regionInfo =
            rfidReader.ReaderCapabilities.SupportedRegions.GetRegionInfo(1);
// regulatoryConfig.Region=regionInfo.RegionCode;
// rfidReader.Config.RegulatoryConfig=regulatoryConfig;
 }
```

Disconnect

When the application is done with the connection and operations on the RFID reader, it calls the following API to close the connection, and to release and clean up the resources.

```
RfidReader.Disconnect();
```

NOTE: If a reader disconnection occurs, the reader.isConnected() flag may return the value false. If the application calls reader.connect(), the application should call reader.disconnect() regardless of the flag status.

Dispose

When the application main activity is destroyed, it is required to dispose the SDK instance so that it can cleanly exit (unregistration and unbind by SDK as required).

```
Readers.Dispose();
```

Knowing the Reader Capabilities

The capabilities (or Read-Only properties) of the reader are known using the ReaderCapabilites class. The reader capabilities include the following:

General Capabilities

- Model Name
- Number of antennas supported.
- Tag Event Reporting Supported - Indicates the reader's ability to report tag visibility state changes (New Tag, Tag Invisible, or Tag Visibility Changed).
- RSSI Filter Supported - Indicates the reader's ability to report tags based on the signal strength of the back-scattered signal from the tag.
- NXP Commands Supported - Indicates whether the reader supports NXP commands such as Change EAS, set Quiet, Reset Quiet, Calibrate.
- Tag LocationingSupported - Indicates the reader's ability to locate a tag.
- DutyCycleValues - List of DutyCycle percentages supported by the reader.

Gen2 Capabilities

- Block Erase - supported
- Block Write - supported
- State Aware Singulation - supported
- Maximum Number of Operation in Access Sequence
- Maximum Pre-filters allowable per antenna
- RF Modes.

Regulatory Capabilities

- Country Code
- Communication Standard.

UHF Band Capabilities

- Transmit Power table
- Hopping enabled
- Frequency Hop table - If hopping is enabled, this table has the frequency information.
- Fixed Frequency table - If hopping is not enabled, this table contains the frequency list used by the reader. The one-based position of a frequency in this list is its channel index.

Reader Identification

Reader ID and Reader ID Type (the reader identification can be MAC or EPC).

```
Console.Out.WriteLine("\nReader ID: " + Reader.ReaderCapabilities.ReaderID.ID);
                    Console.Out.WriteLine("\nModelName: " +
Reader.ReaderCapabilities.ModelName);
                    Console.Out.WriteLine("\nCommunication Standard: " +
                    Reader.ReaderCapabilities.CommunicationStandard.ToString());
                    Console.Out.WriteLine("\nCountry Code: " +
Reader.ReaderCapabilities.CountryCode);
                                        Console.Out.WriteLine("\nRSSI Filter: " +
Reader.ReaderCapabilities.IsRSSIFilterSupported);
                    Console.Out.WriteLine("\nTag Event Reporting: " +
                    Reader.ReaderCapabilities.IsTagEventReportingSupported);
                    Console.Out.WriteLine("\nTag Locating Reporting: " +
Reader.ReaderCapabilities.IsTagLocationingSupported);
                    Console.Out.WriteLine("\nNXP Command Support: " +
                    Reader.ReaderCapabilities.IsNXPCommandSupported);
                    Console.Out.WriteLine("\nBlockEraseSupport: " +
                    Reader.ReaderCapabilities.IsBlockEraseSupported);
                    Console.Out.WriteLine("\nBlockWriteSupport: " +
                    Reader.ReaderCapabilities.IsBlockWriteSupported);
                    Console.Out.WriteLine("\nBlockPermalockSupport:" +
Reader.ReaderCapabilities.IsBlockPermalockSupported);
                    Console.Out.WriteLine("\nRecommisionSupport: "
+Reader.ReaderCapabilities.IsRecommisionSupported);
                    Console.Out.WriteLine("\nWriteWMISupport:" +
Reader.ReaderCapabilities.IsWriteUMISupported);
                    Console.Out.WriteLine("\nRadioPowerControlSupport: "
+Reader.ReaderCapabilities.IsRadioPowerControlSupported);
                    Console.Out.WriteLine("\nHoppingEnabled: " +
Reader.ReaderCapabilities.IsHoppingEnabled);
                    Console.Out.WriteLine("\nStateAwareSingulationCapable: " +
Reader.ReaderCapabilities.IsTagInventoryStateAwareSingulationSupported);
                    Console.Out.WriteLine("\nUTCClockCapable: " +
                    Reader.ReaderCapabilities.IsUTCClockSupported);
                    Console.Out.WriteLine("\nNumOperationsInAccessSequence: " +
                    Reader.ReaderCapabilities.MaxNumOperationsInAccessSequence);
                    Console.Out.WriteLine("\nNumPreFilters: " +
Reader.ReaderCapabilities.MaxNumPreFilters);
                    Console.Out.WriteLine("\nNumAntennaSupported: " +
                    Reader.ReaderCapabilities.NumAntennaSupported);
```

Configuring the Reader

RF Mode

The reader has one or more sets of C1G2 RF Mode that the reader is capable of operating. The supported RF mode is retrieved from the RF Mode table using the ReaderCapabilities class.

The API GetRFModeTableInfo in reader capabilities.RFModes gets the RF Mode table from the reader. The getLinkedProfiles function described below populates the RFModeTable into an List.

```
// The rfModeTable is populated by the GetRFModeTableInfo function
public  RFModeTable rfModeTable = Reader.ReaderCapabilities.RFModes.GetRFModeTableInfo(0);
                       // The linked profiles are added to an list
private void getLinkedProfiles(List<String> linkedProfiles)
        {
            RFModeTableEntry rfModeTableEntry = null;
            for (int i = 0; i < rfModeTable.Length(); i++)
            {
                rfModeTableEntry = rfModeTable.GetRFModeTableEntryInfo(i);
                linkedProfiles.Add(rfModeTableEntry.BdrValue + " " +
                        rfModeTableEntry.Modulation + " " + rfModeTableEntry.PieValue + "
" +
                        rfModeTableEntry.MaxTariValue + " " +
                        rfModeTableEntry.MaxTariValue + " " +
rfModeTableEntry.StepTariValue);
           }
    }
```

Antenna Specific Configuration

The config class contains the Antennas object. The individual antenna is accessed and configured using the index.

Antenna Configuration

The AntennaProperties is used to set the antenna configuration to individual antenna or all the antennas.

The antenna configuration (SetAntennaConfig function) is comprised of Antenna ID, Receive Sensitivity

Index, Transmit Power Index, and Transmit Frequency Index. These indexes refer to the Receive Sensitivity table, Transmit Power table, Frequency Hop table, or Fixed Frequency table, respectively. These tables are available in Reader capabilities. getAntennaConfig function gets the antenna configuration for the given antenna ID.

RF Configuration

The function SetAntennaRFConfig is added to configure antenna RF configuration to individual antenna. This function is similar to SetAntennaConfig but includes additional parameters specific pertaining to the antenna.

The configuration includes Receive Sensitivity Index, Transmit Power Index, Transmit Frequency Index, and RF Mode Table Index. These indexes are refers to the Receive Sensitivity table, Transmit Power table, Frequency Hop table, Fixed Frequency table, or RF Mode table, respectively. These tables are available in Reader capabilities. Also, includes tari, transmit port, receive port and Antenna Stop trigger condition. The stop condition can be 'n' number of attempts, duration based.

The function GetAntennaRfConfig gets the antenna RF configuration for the given antenna ID.

```
Antennas.AntennaRfConfig antennaRfConfig = Reader.Config.Antennas.GetAntennaRfConfig(1);
                    antennaRfConfig.SetrfModeTableIndex(0);
                    antennaRfConfig.Tari=0;
                    antennaRfConfig.TransmitPowerIndex=270;
                    Reader.Config.Antennas.SetAntennaRfConfig(1, antennaRfConfig);
```

Singulation Control

The function getSingulationControl retrieves the current settings of the singulation control from the reader for the given Antenna ID.

To set the singulation control settings, the setSingulationControll method is used. The following settings can be configured:

- Session: Session number to use for inventory operation.
- Tag Population: An estimate of the tag population in view of the RF field of the antenna.
- Tag Transit Time: An estimate of the time a tag typically remains in the RF field.
- State Aware Singulation Action: The action includes the Inventory state and SL flag. The action can be used if only the reader supports this capability. The ReaderCapabilities class helps to determine whether state-aware singulation is supported or not.

Tag Report Configuration

```
// Get Singulation Control for the antenna 1 Antennas.SingulationControl
singulationControl;
                    singulationControl = Reader.Config.Antennas.GetSingulationControl(1);
                  // Set Singulation Control for the antenna 1 Antennas.SingulationControl
singulationControl;
                    singulationControl.setSession(SESSION.SessionS0);
                    singulationControl.setTagPopulation((short)30);
                    singulationControl.Action.setSLFlag(SL_FLAG.SlAll);
          singulationControl.Action.setInventoryState(INVENTORY_STATE.InventoryStateA);
                             Reader.Config.Antennas.SetSingulationControl(1,
         singulationControl);
```

The SDK provides an ability to configure a set of fields to be reported in a response to an operation by a specific active RFID reader.

Supported fields that might be reported include the following:

- First seen time
- Last seen time
- PC value
- RSSI value
- Phase value
- Channel index
- Tag seen count.

The function getTagStorageSettings retrieves the tag report parameters from the reader for the given Antenna ID.

To set the Tag report parameters, the setTagStorageSettings method is used. The following settings can be configured:

```
// Get tag storage settings from the reader
                    TagStorageSettings tagStorageSettings =
Reader.Config.TagStorageSettings;
                    // set tag storage settings on the reader with all fields
                    tagStorageSettings.SetTagFields(TAG_FIELD.AllTagFields);
                    Reader.Config.TagStorageSettings=tagStorageSettings;
```

Regulatory Configuration

The SDK supports managing of regulatory related parameters of a specific active RFID reader.

Regulatory configuration includes the following:

- Code of selected region
- Hopping
- Set of enabled channels.

A set of enabled channels includes only such channels that are supported in the selected region. If hopping configuration is not allowed for the selected regions, a set of enabled channels is not specified.

Regulatory parameters could be retrieved and set via getRegulatoryConfig and setRegulatoryConfig API functions accordingly. The region information is retrieved using getRegionInfo API. The following example demonstrates retrieving of current regulatory settings and configuring the RFID reader to operate in one of supported regions.

```
// Get and Set regulatory configuration settings
                    RegulatoryConfig regulatoryConfig = Reader.Config.RegulatoryConfig;
                    RegionInfo regionInfo =
Reader.ReaderCapabilities.SupportedRegions.GetRegionInfo(1);
                    regulatoryConfig.Region=regionInfo.RegionCode;
          regulatoryConfig.SetIsHoppingOn(regionInfo.IsHoppingConfigurable);
regulatoryConfig.SetEnabledChannels(regionInfo.GetSupportedChannels());
                    Reader.Config.RegulatoryConfig=regulatoryConfig;
```

Saving Configuration

Various parameters of a specific RFID reader configured via SDK are lost after the next power down. The SDK provides an ability to save a persistent configuration of RFID reader. The saveConfig API function can be used to make the current configuration persistent over power down and power up cycles. The following example demonstrates utilization of mentioned API functions.

Saving the configuration

```
Reader.Config.SaveConfig();
```

Reset Configuration to Factory Defaults

The SDK provides a way to reset the RFID reader to the factory default settings. The resetFactoryDefaults API can be used to attain this functionality. Once this method is called, all the reader settings like events, singulation control, etc will be lost and the RFID reader reboots. A connected application shall lose connectivity to the reader and must connect back again and is required to redo the basic steps for initializing the reader. The following example demonstrates utilization of mentioned API function.

Resetting the configuration

```
Reader.Config.SaveConfig();
```

Managing Events

The Application can register for one or more events so as to be notified of the same when it occurs. There are two main events.

- eventReadNotify - Notifies whenever read tag event occurs with read tag data as argument. By default, the event comes with tag data. If not required, disable using function setAttachTagDataWithReadEvent.
- eventStatusNotify - Notifies whenever status event occurs with status event data as argument.

Table 2 Events

Xamarin Events	Description
GpiEvent	Not supported in Android RFID SDK.
BufferFullWarningEvent	When the internal buffers are 90% full, this event is signaled
AntennaEvent	Not supported in Android RFID SDK.
InventoryStartEvent	Inventory Operation started. In case of periodic trigger, this event is triggered for each period.
InventoryStopEvent	Inventory Operation has stopped. In case of periodic trigger this event is triggered for each period.
AccessStartEvent	Not supported in Android RFID SDK.
AccessStopEvent	Not supported in Android RFID SDK.
DisconnectionEvent	Event notifying disconnection from the Reader. The Application can call reconnect method periodically to attempt reconnection or call disconnect method to cleanup and exit.

Table 2 Events

Xamarin Events	Description
BufferFullEvent	When the internal buffers are 100% full, this event is signaled and tags are discarded in FIFO manner.
NxpEasAlarmEvent	Not Supported in Android RFID SDK.
ReaderExceptionEvent	Event notifying that an exception has occurred in the Reader. When this event is signaled, StatusEventData. ReaderExceptionEventData. ReaderExceptionEventType can be called to know the reason for the exception which is coming as part of Events.StatusEventArgs. The Application can continue to use the connection if the reader renders is usable.
HandheldTriggerEvent	A hand-held Gun/Button event Pull/Release has occurred.
TemperatureAlarmEvent	When Temperature reaches Threshold level, this event is generated. The event data contains source name (PA/Ambient),current Temperature and alarm Level (Low, High or Critical)
BatteryEvent	Events notifying different levels of battery, state of the battery, if charging or discharging.
OperationEndSummaryEvent	Event generated when operation end summary has been generated. The data associated with the event contains total rounds, total number of tags and total time in micro secs.
PowerEvent	Events which notify the different power states of the reader device. The event data contains cause, voltage, current and power.

`registering for read tag data notification`

```
EventHandler eventHandler = new EventHandler();
        Reader.Events.AddEventsListener(eventHandler);
        // Subscribe required status notification
        Reader.Events.SetInventoryStartEvent(true);
        myReader.Events.setInventoryStopEvent(true);
        // enables tag read notification. if this is set to false, no tag read notification
        is send
        myReader.Events.SetTagReadEvent(true);
        myReader.Events.SetReaderDisconnectEvent(true);
                                myReader.Events.SetBatteryEvent(true);
```

Read/Status Notify handler

Implement the RfidEventsLister class to receive event notifications

```
class EventHandler : Java.Lang.Object, IRfidEventsListener
    {
    // Read Event Notification
    public void EventReadNotify(RfidReadEvents e)
    {
            // Recommended to use new method getReadTagsEx for better performance in case
of large
            // tag population

            TagData[] myTags = myReader.Actions.GetReadTags(100);
        if (myTags != null)
        {
            for (int index = 0; index < myTags.Length; index++)
            {
                Console.Out.WriteLine("Tag ID " + myTags[index].TagID);
                if (myTags[index].OpCode == ACCESS_OPERATION_CODE.AccessOperationRead
                        &&
                        myTags[index].OpStatus ==
                                ACCESS_OPERATION_STATUS.AccessSuccess)
                {
                    if (myTags[index].MemoryBankData.Length > 0)
                    {
                        Console.Out.WriteLine(" Mem Bank Data " +
                                myTags[index].MemoryBankData);
                    }
                }
            }
        }
    }
    // Status Event Notification
    public void EventStatusNotify(RfidStatusEvents e)
    {
        Console.Out.WriteLine("Status Notification: " +
                e.StatusEventData.StatusEventType);
    }
}
```

Unregistering for read tag data notification

```
Reader.Events.RemoveEventsListener(eventHandler);
```

Device Status Related Events

Device status, such as battery, power, and temperature, is obtained through events after initiating the following API:

```
Reader.Config.GetDeviceStatus(battery, power, temperature);
```

Response to the above API comes as battery event, power event, and temperature event according to the set boolean value in the respective parameters.

The following is an example of how to get these events.

```
try
{
    if (Reader != null)
        Reader.Config.GetDeviceStatus(true, false, false);
    else
        StopTimer();
}
catch (InvalidUsageException e)
{
    e.PrintStackTrace();
}
catch (OperationFailureException e)
{
    e.PrintStackTrace();
}
```

Basic Operations

Tag Storage Settings

This section covers the basic operations that an application needs to perform on an RFID reader which includes inventory and single tag access operations.

The application needs to get the tags from the dll which are reported by the reader. Tags can be reported as part of an Inventory operation (Reader.Actions.Inventory.Perform) or a Read Access operation (Reader.Actions.TagAccess.ReadEvent or Reader.Actions.TagAccess.ReadWait).

Applications can also configure to receive tag reports that indicate the results of access operations as shown below.

```
TagStorageSettings tagStorageSettings = Reader.Config.TagStorageSettings;
Reader.Config.TagStorageSettings=tagStorageSettings;
```

Each tag has a set of associated information along with it. During the Inventory operation, the reader reports theEPC-ID of the tag, where as during the Read-Access operation the requested Memory Bank Data is also reportedapart from EPC-ID. In either case, there is additional information such as PC-bits, RSSI, last time seen, tag seen count, etc. that is available for each tag. This information is reported to the application as TagData for each tagreported by the reader.

Applications can also choose to enable/disable reporting certain fields in TAG_DATA. Disabling certain fields cansometimes improve the performance as the reader and the sdk are not processing that information. It can also resultin specific behavior. For example, disabling reporting an Antenna Id can result in the application receiving a single unique tag even though they were multiple entries of the same tag reported from different antennas. The

following demonstrates enabling the reporting of PeakRSSI, Tag Seen Count, PC and CRC only and disabling other fields such as Antenna ID, Time Stamps, and XPC.

```
TagStorageSettings tagStorageSettings = Reader.Config.TagStorageSettings;
        TAG_FIELD[] tagField = new TAG_FIELD[4];
        tagField[0] = TAG_FIELD.Pc; tagField[1] =
            TAG_FIELD.PeakRssi; tagField[2] =
            TAG_FIELD.TagSeenCount; tagField[3] =
            TAG_FIELD.Crc;
        tagStorageSettings.SetTagFields(tagField);
        Reader.Config.TagStorageSettings=tagStorageSettings;
```

Tag Storage Use Cases

Example use-cases that get tags from the reader are as follows:

Simple Inventory (Continuous)

A Simple Continuous Inventory operation reads all tags in the field of view of all antennas of the connected RFID reader. It uses no filters (pre-filters or post-filters) and the start and stop trigger for the inventory is the default (for example, start immediately when reader.Actions.Inventory.perform is called, and stop immediately when reader.Actions.Inventory.stop is called).

```
// perform simple inventory
        Reader.Actions.Inventory.Perform();
        // Keep getting tags in the eventReadNotify event if registered
        // stop the inventory
    Reader.Actions.Inventory.Stop();
```

Simple Access Operations - On Single Tag

Tag Access operations are performed on a specific tag or applied on tags that match a specific Access-Filter. If no Access-Filter is specified, the Access Operation is performed on all tags in the field of view of chosen antennas.

This section covers the Simple Tag Access operation on a specific tag which is in the field of view of any of the antennas of the connected RFID reader.

```
// dpo should be disabled before any access operation
```

```
Reader.Config.DPOState=DYNAMIC_POWER_OPTIMIZATION.Disable;
```

Read

The application calls method reader.Actions.TagAccess.ReadWait to read data from a specific memory bank.

```
// Read user memory bank for the given tag ID
String tagId = "1234ABCD00000000000025B1";
       TagAccess tagAccess = new TagAccess();
       TagAccess.ReadAccessParams readAccessParams = new
TagAccess.ReadAccessParams(tagAccess);
       TagData readAccessTag;
       readAccessParams.AccessPassword=0;
       readAccessParams.Count=4; // read 4 words
       readAccessParams.MemoryBank=MEMORY_BANK.MemoryBankUser;
       readAccessParams.Offset=0; // start reading from word offset 0
       readAccessTag = Reader.Actions.TagAccess.ReadWait(tagId, readAccessParams, null);
       Console.Out.WriteLine(readAccessTag.MemoryBank.ToString() + " : " +
              readAccessTag.MemoryBankData);
```

Write, Block-Write

The application calls method reader.Actions.TagAccess.WriteWait or reader. Actions. TagAccess. BlockWriteWait to write data to a specific memory bank. The response is returned as a Tagdata from where a number of words can be retrieved.

```
                 // Write user memory bank data
TagData tagData = null;
       String tagId = "1234ABCD00000000000025B1";
       TagAccess tagAccess = new TagAccess();
       TagAccess.WriteAccessParams writeAccessParams = new
TagAccess.WriteAccessParams(tagAccess);
       String writeData = "11223344"; // write data in string
       writeAccessParams.AccessPassword=0;
       writeAccessParams.MemoryBank=MEMORY_BANK.MemoryBankUser;
       writeAccessParams.Offset=0; // start writing from word offset 0
       writeAccessParams.SetWriteData(writeData);
       // antenna Info is null – performs on all antenna
       Reader.Actions.TagAccess.WiteWait(tagId, writeAccessParams, null, tagData);
```

The following shows usage of block write:

NOTE: The same write access parameters are passed as used above to easily switch between two APIs.

```
Reader.Actions.TagAccess.BlockWriteWait(tag Id, writeAccessParams, null, tagData);
```

Lock

The application calls method reader.Actions.TagAccess.lockWait to perform a lock operation on one or more memory banks with specific privileges.

```
                        // Lock the tag
String tagId = "1234ABCD00000000000025B1";
       TagAccess tagAccess = new TagAccess();
       TagAccess.LockAccessParams lockAccessParams = new
TagAccess.LockAccessParams(tagAccess);
       /* lock now */
       lockAccessParams.SetLockPrivilege(LOCK_DATA_FIELD.LockUserMemory,
          LOCK_PRIVILEGE.LockPrivilegeReadWrite);
       lockAccessParams.AccessPassword=0;
              Reader.Actions.TagAccess.LockWait(tagId, lockAccessParams, null);
```

Kill

The application calls method reader.Actions.TagAccess.killWait to kill a tag.

```
                        // Kill the tag
String tagId = "1234ABCD00000000000025B1";
          TagAccess tagAccess = new TagAccess();
          TagAccess.KillAccessParams killAccessParams = new
TagAccess.KillAccessParams(tagAccess);
          killAccessParams.KillPassword=0;
          Reader.Actions.TagAccess.KillWait(tagId, killAccessParams, null);
```

Block Erase

The application calls RFID_BlockErase to erase the contents of a tag.

```
// Block Erase
TagData tagData = new TagData();
       String tagId = "1234ABCD00000000000025B1";
       TagAccess.BlockEraseAccessParams blockEraseAccessParams = new
          TagAccess.BlockEraseAccessParams(tagAccess);
       blockEraseAccessParams.AccessPassword=0;
       blockEraseAccessParams.MemoryBank=(MEMORY_BANK.MemoryBankUser); // user memory bank
       blockEraseAccessParams.Offset=0; // start erasing from word offset 0
       blockEraseAccessParams.Count=8; // number of words to erase
       MConnectedReader.Actions.TagAccess.BlockEraseWait(tagId, blockEraseAccessParams,
       null,
                                             tagData);
```

Block-Permalock

The application calls method Reader.Actions.TagAccess.blockPermalockWait to block a permalock tag. Tags reported as part of Block-Permalock access operation have TagData.getOpCode as AccessOperationBlockPermaLock and TagData.getOpStatus indicating the result of the operation; if TagData.OpStatus is AccessSuccess, TagData.getMemoryBankData contains the Block-Permalock Mask Data.

```
// Block-Perma Lock the tag
String tagId = "1234ABCD00000000000025B1";
        TagAccess tagAccess = new TagAccess();
        TagAccess.BlockPermalockAccessParams blockPermalockAccessParams = new
        TagAccess.BlockPermalockAccessParams(tagAccess);
        byte[] permalockMask = new byte[] { (byte)0xF0, 0x00 };
        blockPermalockAccessParams.ReadLock=true;
        blockPermalockAccessParams.MemoryBank=MEMORY_BANK.MemoryBankUser;
        blockPermalockAccessParams.Offset=0; // start BlockPermalock from word offset 0
        blockPermalockAccessParams.Count=1; // start BlockPermalock from word offset 0
        blockPermalockAccessParams.SetMask(permalockMask);
        blockPermalockAccessParams.MaskLength=2;
            Reader.Actions.TagAccess.BlockPermalockWait(tagId, blockPermalockAccessParams,
            null);
```

Synchronous Access Operation API Updates

All synchronous access operation APIs have been overloaded with new parameters for prefilter support.

APIs: ReadWait, WriteWait, KillWait, LockWait, bBockWriteWait, BlockEraseWait and BlockPermalockWait

```
public TagData ReadWait(Java.Lang.String tagID,
                        TagAccess.ReadAccessParams readAccessParams,
                                    AntennaInfo antennaInfo, bool bPrefilter)
```

Example:

```
Reader.Actions.TagAccess.ReadWait("2F2203447334C3100002EA55", readAccessParams, null,
                                  true);
```

writeWait and blockWriteWait APIs have additional parameter for specifying whether to use TID memory bank as prefilter.

```
WriteWait(Java.Lang.String tagID,
          TagAccess.WriteAccessParams writeAccessParams,
          AntennaInfo antennaInfo,
          TagData tagData,

          bool bPrefilter, bool bTIDPrefilter)
```

Example:
Apply EPC memory bank as prefilter:

```
Reader.Actions.TagAccess.WriteWait(tagId, writeAccessParams, null, tagData, true, false);
```

Apply TID memory bank as prefilter:

```
Reader.Actions.TagAccess.WriteWait(tagId, writeAccessParams, null, tagData, true, true);
```

Provision of TID memory bank as prefilter and giving number of retries will help doing retries on partial writes of EPC memory bank.

NOTE: Using TID memory bank as prefilter will result in longer time of write, as SDK internally retrieves TID memory bank of tag.

NOTE: When prefilter is enabled, SDK does access operation in session S0 and uses InvBNotInvAOrDsrtSlNotAsrtSl with InvB inventory state

TagAccess.WriteAccessParams have additional parameter to define number of retries.

```
WriteRetries=int writeRetries;
```

Number of retries to be performed for write access operation.
In case of failure SDK retries to perform write operation from data offset from last successful number of words written.

API to configure access operation time out via Config class.

Earlier version of SDK has hardcoded timeout of five seconds which results in longer wait for operation result to come if tag is not found in FOV or other reasons (not singulated). By this application can shorten wait time and quickly retry in failure cases.

```
AccessOperationWaitTimeout=int timeout
```

Method to set access operation timeout for all synchronous tag access operations APIs.

Example:

```
Reader.Config.AccessOperationWaitTimeout=1000;
```

Advanced Operations

Using Pre-Filters

Pre-filters are the same as the Select command of C1G2 specification. Once applied, pre-filters are applied prior to Inventory and Access operations.

Introduction

Singulation

Singulation refers to the method of identifying an individual Tag in a multiple-Tag environment. RFID readers support State-Aware or State-Unaware pre-filtering (or singulation) which is indicated by the boolean flag IsTagInventoryStateAwareSingulationSupported in the ReaderCapabilities class.

In order to filter tags that match a specific condition, it is necessary to use the tag-sessions and their states (setting the tags to different states based on match criteria -`Reader.Actions.PreFilters.Add`) so that while performing inventory, tags can be instructed to participate (singulation `Reader.Config.Antennas.SingulationControl`) or not participate in the inventory based on their states.

Sessions and Inventoried Flags

Tags provide four sessions (denoted S0, S1, S2, and S3) and maintain an independent inventoried flag for each session. Each of the four inventoried flags has two values, denoted A and B. These inventoried flag of each session can be set to A or B based on match criteria using method `Reader.Actions.PreFilters.Add`.

Selected Flag

Tags provide a selected flag, SL, which is asserted or deasserted based on match criteria using method `Reader.Actions.PreFilters.Add`.

State-Aware Singulation

In state-aware singulation, the application specifies detailed controls for singulation: Action and Target. Action indicates whether matching Tags assert or de-assert SL (Selected Flag), or set their inventoried flag to A or to B. Tags conforming to the match criteria specified using the method `Reader.Actions.PreFilters.Add` are considered matching and the remaining are non-matching. Target indicates whether to modify a tag's SL flag or its inventoried flag, and in the case of inventoried, it further specifies one of four sessions.

Applying Pre-Filters

The following are the steps to use pre-filters:

1. Add pre-filters.
2. Set appropriate singulation controls.
3. Perform Inventory or Access operation.

Add Pre-filters

Each RFID reader supports a maximum number of pre-filters per antenna as indicated by `ReaderCapabilities.MaxNumPreFilters` which is known using the ReaderCapabilities.
The application sets pre-filters using reader.Actions.PreFilters.add and removes using `Reader.Actions.PreFilters.Delete`.

State-Aware Settings

Add state aware pre-filter

```
PreFilters filters = new PreFilters();
            PreFilters.PreFilter filter = new PreFilters.PreFilter(filters);
            byte[] tagMask = new byte[] { 0x12, 0x11 };
            filter.AntennaID=(short)1;// Set this filter for Antenna ID 1
            filter.SetTagPattern(tagMask);// Tags which starts with 0x1211
            filter.TagPatternBitCount=tagMask.Length * 8;
            filter.BitOffset=32; // skip PC bits (always it should be in bit length)
            filter.MemoryBank=MEMORY_BANK.MemoryBankEpc;
            filter.FilterAction=FILTER_ACTION.FilterActionStateAware;
 // use state aware singulation
            filter.StateAwareAction.Target=TARGET.TargetInventoriedStateS1;
// inventoried flag of session
 // S1 of matching tags to B

filter.StateAwareAction.StateAwareAction=STATE_AWARE_ACTION.StateAwareActionInvB;
            // not to select tags that match the criteria
            Reader.Actions.PreFilters.Add(filter);
            // It is also required to set appropriate singulation control not to
            // get tags with inventoried flag B for session 1
```

Set Appropriate Singulation Controls

Now that the pre-filters are set (for example, tags are classified into matching or non-matching criteria), the application needs to specify which tags participate in the inventory using `Reader.Config.Antennas.SingulationControl` Singulation Control must be specified with respect to each antenna such as pre-filters.

State-Aware Singulation

Set the singulation control

```
Antennas.SingulationControl s1_singulationControl =
                        rfidReader.Config.Antennas.GetSingulationControl(1);
              s1_singulationControl.Session=SESSION.SessionS1;
s1_singulationControl.Action.InventoryState=INVENTORY_STATE.InventoryStateB;
              s1_singulationControl.Action.SLFlag=SL_FLAG.SlFlagDeasserted;
s1_singulationControl.Action.SetPerformStateAwareSingulationAction(true);
                         rfidReader.Config.Antennas.SetSingulationControl(1,
           s1_singulationControl);
```

Perform Inventory or Access operation

Inventory or Access operation when performed after setting pre-filters, use the tags filtered out of pre-filters for their operation.

Using Triggers

Triggers are the conditions that are satisfied in order to start or stop an operation (Inventory or Access Sequence). This information can be specified using TriggerInfo class. The application can also configure the Tag-Report trigger which indicates when to receive 'n' unique Tag-Reports from the Reader.

We have to use `Config.StartTrigger` and `Config.StopTrigger` APIs to set triggers on the reader.

The following are some use-cases of using TRIGGER_INFO:

- Periodic Inventory: Start inventory at a specified time for a specified duration repeatedly.

```
TriggerInfo triggerInfo = new TriggerInfo();
              // start inventory at every 2 seconds
triggerInfo.StartTrigger.setTriggerType(START_TRIGGER_TYPE.START_TRIGGER_TYPE_PERIODIC);
              triggerInfo.StartTrigger.Periodic.Period=2000; // perform inventory for 2
seconds
                                                                      // stop trigger
triggerInfo.StopTrigger.setTriggerType(STOP_TRIGGER_TYPE.STOP_TRIGGER_TYPE_DURATION);
                              triggerInfo.StopTrigger.DurationMilliSeconds=200; // stop after
               200 milliseconds
```

- Perform ‘n’ Rounds of Inventory with a timeout: Start condition can be any; Stop condition is to perform ‘n’ rounds of inventory and then stop or stop inventory after the specified timeout.

```
 TriggerInfo triggerInfo = new TriggerInfo();
// start inventory immediate
triggerInfo.StartTrigger.TriggerType=START_TRIGGER_TYPE.StartTrigger
TypeImmediate;
// stop trigger
triggerInfo.StopTrigger.TriggerType=STOP_TRIGGER_TYPE.StopTriggerTypeAccessNAttemptsWithT
imeout;
        triggerInfo.StopTrigger.NumAttempts.SetN((short)3); // perform 3 rounds of
inventory
         triggerInfo.StopTrigger.NumAttempts.Timeout=3000; // timeout after 3 seconds
         //
```

- Read ‘n’ tags with a timeout: Start condition could be any; Stop condition is to stop after reading ‘n’ tags or stop inventory after the specified timeout.

```
TriggerInfo triggerInfo = new TriggerInfo();
        // start inventory immediate
    triggerInfo.StartTrigger.TriggerType=START_TRIGGER_TYPE.StartTrigger
    TypeImmediate;
        // stop trigger
triggerInfo.StopTrigger.TriggerType=STOP_TRIGGER_TYPE.StopTriggerTypeTagObservationWithTi
meout;
        triggerInfo.StopTrigger.TagObservation.SetN((short)100); // stop inventory
after reading 100 tags
       triggerInfo.StopTrigger.TagObservation.Timeout=3000; // timeout after 3 seconds
               // report back all read tags after getting 100 unique tags or after
               3 seconds
```

- Inventory based on hand-held trigger: Start inventory when the hand-held gun/button trigger is pulled, and stop inventory when the hand-held gun/button trigger is released or subject to timeout.

```
TriggerInfo triggerInfo = new TriggerInfo();
triggerInfo.StartTrigger.TriggerType=START_TRIGGER_TYPE.StartTriggerTypeHandheld;
        // Start Inventory when the Handheld trigger is pressed
        triggerInfo.StartTrigger.Handheld.HandheldTriggerEvent=
HANDHELD_TRIGGER_EVENT_TYPE.HandheldTriggerPressed;
        triggerInfo.StopTrigger.TriggerType=STOP_TRIGGER_TYPE.StopTriggerTypeDuration;
        // Stop Inventory when the Handheld trigger is released
        triggerInfo.StopTrigger.Handheld.HandheldTriggerEvent =
HANDHELD_TRIGGER_EVENT_TYPE.HandheldTriggerReleased;
        triggerInfo.StopTrigger.Handheld.HandheldTriggerTimeout=0;
```

- Set the trigger using the following APIs, perform inventory and other operations which are using above set start and stop triggers.

```
Reader.Config.StartTrigger=triggerInfo.StartTrigger;
Reader.Config.StopTrigger=triggerInfo.StopTrigger;
Reader.Actions.Inventory.Perform();
```

Access

Using Access-Filters

In order to perform an access operation on multiple tags, the application can set ACCESS_FILTER to filter the required tags. If ACCESS_FILTER is not specified, the operation is performed on all tags. In any case, the PRE_FILTER(s) (if any is set) applies prior to ACCESS_FILTER.

The following access-filter gets all tags that have zeroed reserved memory bank.

```
 AccessFilter accessFilter = new AccessFilter();
            byte[] tagMask = new byte[] { (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff,
(byte)0xff,
                    (byte)0xff, (byte)0xff, (byte)0xff };
            // Tag Pattern A
            accessFilter.TagPatternA.MemoryBank=MEMORY_BANK.MemoryBankReserved;
            accessFilter.TagPatternA.SetTagPattern(new byte[] { 0x00, 0x00, 0x00, 0x00,
0x00, 0x00,
                    0x00, 0x00 }); accessFilter.TagPatternA.TagPatternBitCount=(8 * 8);
            accessFilter.TagPatternA.BitOffset=0;
            accessFilter.TagPatternA.SetTagMask(tagMask);
            accessFilter.TagPatternA.TagMaskBitCount=tagMask.Length * 8;
            accessFilter.AccessFilterMatchPattern=FILTER_MATCH_PATTERN.A;
```

Access Operation on Multiple Tags

Performing a single Access operation on multiple tags is an asynchronous operation. The function issues the access-operation and returns. The reader performs one round of inventory using pre-filters, if any, and then applies the access-filters and the resultant tags are subject to the access-operation. When the access operation is complete, the SDK signals the eventStatusNotify event with event data as `InventoryStopEvent`.

In case of Read access operation (`Reader.Actions.TagAccess.ReadEvent`) the event eventReadNotify is signaled when tags are reported.

The following demonstrates a sample write-access operation: //
Create Event to signify access operation complete.

```
 Reader.Events.SetInventoryStartEvent(true);
        Reader.Events.SetInventoryStopEvent(true); // Data Read Notification from the
reader
class EventHandler : Java.Lang.Object, IRfidEventsListener
    {
    // Read Event Notification
    public void EventReadNotify(RfidReadEvents e)
    {
        TagData tag = e.ReadEventData.TagData;
        Console.Out.WriteLine("Tag ID " + tag.TagID);
        if (tag.OpCode == ACCESS_OPERATION_CODE.AccessOperationRead &&
                tag.OpStatus == ACCESS_OPERATION_STATUS.AccessSuccess)
        {
            if (tag.MemoryBankData.Length > 0)
            {
                Console.Out.WriteLine(" Mem Bank Data " + tag.MemoryBankData);
            }
        }
    }
    // Status Event Notification
    public void EventStatusNotify(RfidStatusEvents e){
        if (e.StatusEventData.StatusEventType ==
                STATUS_EVENT_TYPE.InventoryStartEvent)
        {
            // Access operation started
        }
        else if (e.StatusEventData.StatusEventType ==
              STATUS_EVENT_TYPE.InventoryStopEvent)
        {
            // Access operation stopped - Can be used to signal waiting thread
        }
    }
}
```

```
// Access Filter - EPC ID starting with 0x1122
AccessFilter accessFilter = new AccessFilter();
            byte[] tagMask = new byte[] { 0xff, 0xff };
            // Tag Pattern A
            accessFilter.TagPatternA.MemoryBank=MEMORY_BANK.MemoryBankEpc;
            accessFilter.TagPatternA.SetTagPattern(new byte[] { 0x11, 0x22 });
            accessFilter.TagPatternA.TagPatternBitCount=2 * 8;
            accessFilter.TagPatternA.BitOffset=0;
            accessFilter.TagPatternA.SetTagMask(tagMask);
            accessFilter.TagPatternA.TagMaskBitCount=tagMask.Length * 8;
            accessFilter.AccessFilterMatchPattern=FILTER_MATCH_PATTERN.A;

            // Write user memory bank data
            TagAccess tagAccess = new TagAccess();
            TagAccess.WriteAccessParams writeAccessParams = new
TagAccess.WriteAccessParams(tagAccess);
            String writeData = "ABCDABCD";
            writeAccessParams.AccessPassword=0;
            writeAccessParams.MemoryBank=MEMORY_BANK.MemoryBankUser;
            writeAccessParams.Offset=0; // start writing from word offset 0
            writeAccessParams.SetWriteData(writeData);
            // Asynchronous write operation
            Reader.Actions.TagAccess.WriteEvent(writeAccessParams, accessFilter, null);
            // wait for access operation to complete (INVENTORY_STOP_EVENT is signaled
after completing
            //the access operation in the eventStatusNotify)
```

Using Access Sequence

The application issues multiple access operations on a single go using Access-Sequence API. This is useful when each tag from a set of (access-filtered) tags is to be subject to an order of access operations.

The maximum number of access-operations that can be specified in an access sequence is available in `rfidReader.ReaderCapabilities.MaxNumOperationsInAccessSequence` of ReaderCapabilities class.

The operations are performed in the same order in which it is added to it sequence. An operation can be removed from the sequence using `reader.Actions.TagAccess.OperationSequence.Delete` and finally de-initialized if no longer needed by calling the function.

```
 Reader.Actions.TagAccess.OperationSequence.DeleteAll();
// add Write Access operation - Write to User memory
                   TagAccess tagAccess = new TagAccess();
           TagAccess.Sequence opSequence = new TagAccess.Sequence(tagAccess, tagAccess);
           TagAccess.Sequence.Operation op1 = new
TagAccess.Sequence.Operation(opSequence);
           op1.AccessOperationCode=ACCESS_OPERATION_CODE.AccessOperationWrite;
           op1.WriteAccessParams.MemoryBank=MEMORY_BANK.MemoryBankUser;
           op1.WriteAccessParams.AccessPassword=0;
           op1.WriteAccessParams.Offset=0;
           op1.WriteAccessParams.SetWriteData("55667788" );
       op1.WriteAccessParams.WriteDataLength=4;
           Reader.Actions.TagAccess.OperationSequence.Add(op1);
// add Write Access operation - Write to Reserved memory bank
       TagAccess.Sequence.Operation op2 = new TagAccess.Sequence.Operation(opSequence);
       op2.AccessOperationCode=ACCESS_OPERATION_CODE.AccessOperationWrite;
       op2.WriteAccessParams.MemoryBank=MEMORY_BANK.MemoryBankUser;
       op2.WriteAccessParams.AccessPassword=0;
       op2.WriteAccessParams.Offset=0;
       op2.WriteAccessParams.SetWriteData("BBBBCCCC");
op2.WriteAccessParams.WriteDataLength=4;
           Reader.Actions.TagAccess.OperationSequence.Add(op2);
           // perform access sequence

           Reader.Actions.TagAccess.OperationSequence.PerformSequence();

          // if the access operation is to be terminated without meeting stop trigger (if
specified),

           // stopSequence method can be called

           Reader.Actions.TagAccess.OperationSequence.StopSequence();
```

Gen2v2 Operations

This section covers the Gen2V2 operations that an application needs to performed on a RFID Reader which supports Gen2v2 commands such as authenticate, untraceable, and readbuffer.

Authenticate

Authenticate operation takes in the message data and message length with few of the options such as decision on including the response length, sending the response, etc.

The AuthenticateParams contain the message data, message length and other settings to be sent to the reader. The accessfilter parameter contains the tag pattern on which the operation occurs.

```
// authenticate
// Tag Pattern A
AccessFilter accessFilter = null;
           accessFilter.TagPatternA.MemoryBank=MEMORY_BANK.MemoryBankEpc;
           accessFilter.TagPatternA.SetTagPattern(new byte[] { (byte)0xe2, (byte)0xc0 });
           accessFilter.TagPatternA.TagPatternBitCount=16;
           accessFilter.TagPatternA.BitOffset=32;
           accessFilter.TagPatternA.SetTagMask(tagMask);
           accessFilter.TagPatternA.TagMaskBitCount=tagMask.Length * 8;
              accessFilter.AccessFilterMatchPattern=(FILTER_MATCH_PATTERN.A);
```

```
// G2V2 authenticate
// Gen2V2 gen2V2 - new Gen2v2 ();
 Gen2v2.AuthenticateParams AuthenticateParams = new Gen2v2.AuthenticateParams(gen2V2);
            AuthenticateParams.MsgData="2001FD5D8048F48DD09AAD22000111";
            AuthenticateParams.MsgLen=120;
            AuthenticateParams.IncrespLen=true;
            AuthenticateParams.StoreResp=false;
            AuthenticateParams.SentResp=true;
            try
            {
               Reader.Actions.Gen2v2Access.Authenticate(AuthenticateParams, accessFilter,
null);
            }
            catch (InvalidUsageException e)
            {
                e.PrintStackTrace();
            }
            catch (OperationFailureException e)
            {
                e.PrintStackTrace();
                   }
// Keep getting response in the eventReadNotify event if registered
```

The response and result in the Tagdata will contain the information obtained from the operation.

```
 public class EventHandler : Java.Lang.Object, IRfidEventsListener
    {
    // Read Event Notification
    public void EventReadNotify(RfidReadEvents e)
    {
        TagData[] myTags = reader.Actions.GetReadTags(100);
        if (myTags != null)
        {
            for (int index = 0; index < myTags.Length; index++)
            {
                Console.Out.WriteLine("Tag ID " + myTags[index].TagID);
                if ((myTags[index].G2v2OpStatus != null) &&
                        (myTags[index].G2v2OpStatus ==
                                GEN2V2_OPERATION_STATUS.AccessSuccess))
                {
                    if (myTags[index].G2v2Response.Any())
                    {
                        Console.Out.WriteLine("Gen2v2 authenticate response " +
                                myTags[index].G2v2Response);
                    }
                }
            }
        }
    }// Status Event Notification
    public void EventStatusNotify(RfidStatusEvents e)
    {
        Console.Out.WriteLine("Status Notification: " +
                e.StatusEventData.StatusEventType);
        if (e.StatusEventData.StatusEventType == STATUS_EVENT_TYPE.InventoryStartEvent)
        {
            // Access operation started
        }
        else if (e.StatusEventData.StatusEventType ==
               STATUS_EVENT_TYPE.InventoryStopEvent)
        {
            // Access operation stopped - Can be used to signal waiting thread
        }
    }
}
```

Untraceable

Untraceable operation lets the user decide which memory bank to show and what length of the memory bank to show. Here the UntraceableParams contain the settings and password. The accessfilter parameter contains the tag pattern on which the operation occurs.

```
 // untraceable
            AccessFilter accessFilter = new AccessFilter();
            byte[] tagMask = new byte[] { (byte)0xff, (byte)0xff, };
            // Tag Pattern A
            accessFilter.TagPatternA.MemoryBank=MEMORY_BANK.MemoryBankEpc;
            accessFilter.TagPatternA.SetTagPattern("2f22");
            accessFilter.TagPatternA.TagPatternBitCount=32;
            accessFilter.TagPatternA.BitOffset=32;
            accessFilter.TagPatternA.SetTagMask(tagMask);
            accessFilter.TagPatternA.TagMaskBitCount=tagMask.Length * 8;
            accessFilter.AccessFilterMatchPattern=FILTER_MATCH_PATTERN.A;
            Gen2v2 gen2V2 = new Gen2v2();
            Gen2v2.UntraceableParams UntraceableParams = new
Gen2v2.UntraceableParams(gen2V2);
            UntraceableParams.Password=0;
            UntraceableParams.ShowEpc=true;
            UntraceableParams.HideEpc=false;
            UntraceableParams.ShowUser=false;
            UntraceableParams.EpcLen=6;
            UntraceableParams.Tid=UNTRACEABLE_TID.HideAllTid;
            try
            {
                Reader.Actions.Gen2v2Access.Untraceable(UntraceableParams, accessFilter,
null);
            }
            catch (InvalidUsageException e)
            {
                e.PrintStackTrace();
            }
            catch (OperationFailureException e)
            {
                e.PrintStackTrace();
                   }
```

After this, when inventory is run, the effects of the settings sent are seen in untraceable operation.

Resetting the Reader

The RFID SDK supports performing soft-reset of the reader. A connected application shall lose connectivity to the reader, must connect back again, and is required to redo the basic steps for initializing the reader. The following example demonstrates utilization of then API function.

```
// Resetting the reader
Reader.Actions.Reset();
```

Tag Locationing

Readers that support the Tag Locationing feature report the same in the field isTagLocationingSupported of ReaderCapabilities as true. This feature is supported only on hand-held readers and is useful to locate a specific tag in the field of view of the reader's antenna. The default locationing algorithm supported on the reader can perform locationing only on a single antenna. reader.Actions.TagLocationing.Perform can be used to start locating a tag, and reader.Actions.TagLocationing.Stop to stop the locationing operation. The result of locationing of a tag is reported as LocationInfo in TagData and is present in TagData if tagData isContainsLocationInfo is true. tagData.LocationInfo.getRelativeDistance gives the relative distance of the tag from the reader antenna.

```
// Performing Tag Locationing on a particular tag ID
Reader.Actions.TagLocationing.Perform("E2002849491502421020B330", null, null);
            try
            {
                Thread.Sleep(5000);
            }
            catch (Java.Lang.InterruptedException e)
            {
                e.PrintStackTrace();
            }
            Reader.Actions.Inventory.Stop();
            //The response of the tag locationing comes through eventReadNotify in the
following
            Event Handler;
public class EventHandler : Java.Lang.Object, IRfidEventsListener
        {
// Read Event Notification
public void EventReadNotify(RfidReadEvents e)
        {
            TagData[] myTags = Reader.Actions.GetReadTags(100);
            if (myTags != null)
            {
                for (int index = 0; index < myTags.Length; index++)
                {
                    Console.Out.WriteLine("Tag ID " + myTags[index].TagID);
                    if (myTags[index].IsContainsLocationInfo)
                    {
                      int tag = index; Console.Out.WriteLine("Tag locationing distance " +
                        myTags[tag].LocationInfo.RelativeDistance);
                    }
                }
            }
          }
      }
```

Trigger Mode - RFID and Barcode

To set the trigger mode to work as RFID or Barcode functionality, use the following API.

```
rfidReader.Config.TriggerMode=(ENUM_TRIGGER_MODE.RfidMode, true);
```

First parameter is mode enum value and second parameter indicates whether SDK should take care of disabling scanner plugin. When second parameter is true; SDK executes code to disable scanner plugin.

NOTE: It is recommended that Application handle scanner plugin enable/disable in pause and resume activity respectively to avoid cross triggering to functionality not required.

Refer to the DataWedge documentation at the following locations:

techdocs.zebra.com/datawedge/6-5/guide/api/scannerinputplugin/

techdocs.zebra.com/datawedge/6-5/guide/api/resultinfo/

The following code demonstrates ways to disable scanner plugin.

```
private void EnableDisableScannerPlugIn(bool enable)
        {
            // define action and data strings
            String scannerInputPlugin = "com.symbol.datawedge.api.ACTION";
            String extraData = "com.symbol.datawedge.api.SCANNER_INPUT_PLUGIN";
            // following flag is updated in RESULT INFO received for scanner status
            mScannerStatusReceived = false;
            if (enable)
            {
                // enable scanner plugin
                // create the intent
                Intent i = new Intent();
                // set the action to perform
                i.SetAction(scannerInputPlugin);
                // add additional info
                i.PutExtra(extraData, "ENABLE_PLUGIN");
                //
                i.PutExtra("SEND_RESULT", "true");
                i.PutExtra("COMMAND_IDENTIFIER", "RFIDDEMO_ENABLE_SCANNER");
                // send the intent to DataWedge
                this.sendBroadcast(i);
            }
            else
            {
                //disable scanner plugin
                // create the intent
                Intent i = new Intent();
                // set the action to perform
                i.SetAction(scannerInputPlugin);
                // add additional info
                i.PutExtra(extraData, "DISABLE_PLUGIN");
                //
                i.PutExtra("SEND_RESULT", "true");
                i.PutExtra("COMMAND_IDENTIFIER", "RFIDEMO_DISABLE_SCANNER");
                // send the intent to DataWedge
                this.sendBroadcast(i);
            }
            // wait for synchronization response
                int timeout = 0;
while (timeout++ < 10 && !mScannerStatusReceived)
            {
                try
                {
                    Thread.Sleep(100);
                }
                catch (Java.Lang.InterruptedException e)
                {
                    e.PrintStackTrace();
                }
            }
            Log.Debug(TAG, "setTriggerMode synchronization done");
```

Set Attribute

The RFID SDK supports setting RFID parameters through attributes.

The following code demonstrates ways to set reader LED Mode indication.

```
SetAttribute setAttributeinfo = new SetAttribute();
setAttributeinfo.Attvalue="2";
setAttributeinfo.Atttype="B";
setAttributeinfo.Attnum=1785;
rfidReader.Config.SetAttribute(setAttributeinfo);
```

Set Host LED Support

The RFID SDK supports setting host LED indication for tag read events using following APIs:

```
Reader.Config.SetLedBlinkEnable(true);
```

Set Default Configuration

This API helps to rapidly set default configuration on the reader.

API supports the following reader configurations:

- Antenna configuration
- Singulation settings
- Tag storage settings
- Delete all pre-filters (select record) on device
- Dynamic power setting (DPO enable/disable)
- Any five attributes passed as array of attributes

- API sets start and stop trigger type to immediate by itself.

```
// antenna power
Antennas.AntennaRfConfig antennaRfConfig =
            mConnectedReader.Config.Antennas.GetAntennaRfConfig(1);
            //mConnectedReader.Config.Antennas.new AntennaRfConfig();
                                antennaRfConfig.TransmitPowerIndex=270;

// singulation control
Antennas.SingulationControl singulationControl =
            mConnectedReader.Config.Antennas.GetSingulationControl(1);
            singulationControl.Action.InventoryState=INVENTORY_STATE.InventoryStateA;
               singulationControl.Action.SetPerformStateAwareSingulationAction(false);

// Tag storage settings

TagStorageSettings tagStorageSettings = new TagStorageSettings();
            TAG_FIELD[] tagfields = new TAG_FIELD[2];
            tagfields[0] = TAG_FIELD.PeakRssi;
            tagfields[1] = TAG_FIELD.TagSeenCount;
                              tagStorageSettings.SetTagFields(tagfields);

// following is attribute to set trigger mode as RFID
SetAttribute[] setAttributeArray = new SetAttribute[1];
setAttributeArray[0] = new SetAttribute(1644,"B","0",0);

// API call


mConnectedReader.Config.SetDefaultConfigurations(antennaRfConfig, singulationControl,
         tagStorageSettings, true, true, setAttributeArray);
```

Exceptions

The Zebra RFID Android SDK throws two types of exceptions as a given:

- InvalidUsageException: This exception is thrown when the user passes an invalid parameter, calling getInfo() gives detail error message.
- OperationFailureException: This exception is thrown when the requested operation is failed. The Exception contains the Operation RFIDResults, status description, time stamp & vendor specific message for the operation failure.

Exception Handling

All API should be called under try-catch block to catch the exception thrown while performing API by SDK.

```
try
            {
                Reader.Connect();
            }
            catch (InvalidUsageException e)
            {
                e.PrintStackTrace();
            }
            catch (OperationFailureException e)
            {
                e.PrintStackTrace();
      }
```

General Guidelines

Synchronization

Use synchronization method to cover connection, configuration, and disconnection blocks.

Connection related APIs:

```
lock (monitor)
                {
                    readers.AvailableRFIDReaderList;
                    Reader.Connect();
                    // Code related to configuration and initial setup after connection
                    ConfigureRfidReader();
                }
```

Disconnection related APIs:

```
lock (monitor)
                {
                    Reader.Disconnect();
                    Readers.Dispose();
                }
```

Threading

The following APIs must be called from background thread:

- `GetAvailableRFIDReaderList`
- connect
- disconnect
- Dispose

Quick Start Sample

This code provides quick implementation of MainActivity with RFID SDK and inventory on hand-held trigger.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Android.App;
using Android.Content;
using Android.OS;
using Android.Runtime;
using Android.Support.V7.App;
using Android.Views;
using Android.Widget;
using Android.Util;
using Com.Zebra.Rfid.Api3;
using Java.Util;
using System.Threading;

namespace XamarinApiAutomation
{
    [Activity(Label = "MainActivity", MainLauncher = true)]
    public class MainActivity : AppCompatActivity
    {
        private static Readers readers;
        private static IList<ReaderDevice> availableRFIDReaderList;
        private static ReaderDevice readerDevice;
        private static RFIDReader Reader;
        private static String TAG = "DEMO";
        TextView textView;
        private EventHandler eventHandler;
        protected override void OnCreate(Bundle savedInstanceState)
        {
            base.OnCreate(savedInstanceState);
            SetContentView(Resource.Layout.list_testcases);
            // Create your application here
            // textView = (TextView)FindViewById(Resource.Id.TagText);
            // SDK
            if (readers == null)
            {
                readers = new Readers(this, ENUM_TRANSPORT.ServiceSerial);
            }

            ThreadPool.QueueUserWorkItem(o =>
            {
                try
```

```
                {
             if (readers != null && readers.AvailableRFIDReaderList != null)
                    {
                            availableRFIDReaderList =
                                    readers.AvailableRFIDReaderList;
                                  if (availableRFIDReaderList.Count > 0)
                                  {
                                      if (Reader == null)
                                      {
                                          // get first reader from list
                                          readerDevice = availableRFIDReaderList[0];
                                Reader = readerDevice.RFIDReader;
                                          // Establish connection to the RFID Reader
                                          Reader.Connect();
                                          if (Reader.IsConnected)
                                          {
                                              Console.Out.WriteLine("Readers connected");
                                        ConfigureReader();
                                          }

                                      }
                                  }
                    }
         }
                catch (InvalidUsageException e)
                {
                    e.PrintStackTrace();
                }
                catch (OperationFailureException e)
                {
                    e.PrintStackTrace();
                    Log.Debug(TAG, "OperationFailureException " + e.VendorMessage);
                }

            });
        }
        private void ConfigureReader()
        {
            if (Reader.IsConnected)
            {
                TriggerInfo triggerInfo = new TriggerInfo();
                triggerInfo.StartTrigger.TriggerType =
START_TRIGGER_TYPE.StartTriggerTypeImmediate;
                triggerInfo.StopTrigger.TriggerType =
STOP_TRIGGER_TYPE.StopTriggerTypeImmediate;
                try
                {
                    // receive events from reader
```

```
                if (eventHandler == null)
                    eventHandler = new EventHandler(Reader);
                Reader.Events.AddEventsListener(eventHandler);
                // HH event
                Reader.Events.SetHandheldEvent(true);
                // tag event with tag data
                Reader.Events.SetTagReadEvent(true);
                Reader.Events.SetAttachTagDataWithReadEvent(false);
                // set trigger mode as rfid so scanner beam will not come
                Reader.Config.SetTriggerMode(ENUM_TRIGGER_MODE.RfidMode, true);
                // set start and stop triggers
                Reader.Config.StartTrigger = triggerInfo.StartTrigger;
                Reader.Config.StopTrigger = triggerInfo.StopTrigger;
            }
            catch (InvalidUsageException e)
            {
                e.PrintStackTrace();
            }
            catch (OperationFailureException e)
            {
                e.PrintStackTrace();
            }
        }
    }
    protected override void OnDestroy()
    {

        try
        {
            if (Reader != null)
            {
                Reader.Events.RemoveEventsListener(eventHandler);
                Reader.Disconnect();
                Toast.MakeText(ApplicationContext, "Disconnecting reader",
                        ToastLength.Long).Show();
                Reader = null;
                readers.Dispose();
                readers = null;
            }
        }
        catch (InvalidUsageException e)
        {
            e.PrintStackTrace();
        }
        catch (OperationFailureException e)
        {
            e.PrintStackTrace();
        }
```

```
            catch (Exception e)
            {
                e.StackTrace.ToString();
            }
        }

        // Read/Status Notify handler
        // Implement the RfidEventsLister class to receive event notifications
        public class EventHandler : Java.Lang.Object, IRfidEventsListener
        {
       public EventHandler (RFIDReader Reader)
       {

       }
            // Read Event Notification
            public void EventReadNotify(RfidReadEvents e)
            {
               // Recommended to use new method getReadTagsEx for better performance in case
of large tag population

                TagData[] myTags = Reader.Actions.GetReadTags(100);
                if (myTags != null)
                {
                    for (int index = 0; index < myTags.Length; index++)
                    {
                        Log.Debug(TAG, "Tag ID " + myTags[index].TagID);
                        if (myTags[index].OpCode ==
                                ACCESS_OPERATION_CODE.AccessOperationRead &&
                                myTags[index].OpStatus ==
                                        ACCESS_OPERATION_STATUS.AccessSuccess)
                        {
                            if (myTags[index].MemoryBankData.Length > 0)
                            {
                                Log.Debug(TAG, " Mem Bank Data " +
myTags[index].MemoryBankData);
                            }
                        }
                    }
                }
            }
            // Status Event Notification
            public void EventStatusNotify(RfidStatusEvents rfidStatusEvents)
            {
                Log.Debug(TAG, "Status Notification: " +
                        rfidStatusEvents.StatusEventData.StatusEventType);
                if (rfidStatusEvents.StatusEventData.StatusEventType ==
                        STATUS_EVENT_TYPE.HandheldTriggerEvent)
                {
```

```
                    if
(rfidStatusEvents.StatusEventData.HandheldTriggerEventData.HandheldEvent ==
                            HANDHELD_TRIGGER_EVENT_TYPE.HandheldTriggerPressed)
                    {
                        ThreadPool.QueueUserWorkItem(o =>
                        {
                            try
                            {
                                Reader.Actions.Inventory.Perform();
                            }
                            catch (InvalidUsageException e)
                            {
                                e.PrintStackTrace();
                            }
                            catch (OperationFailureException e)
                            {
                                e.PrintStackTrace();
                            }
                        });

                    }
                    if
(rfidStatusEvents.StatusEventData.HandheldTriggerEventData.HandheldEvent ==
                            HANDHELD_TRIGGER_EVENT_TYPE.HandheldTriggerReleased)
                    {
                        ThreadPool.QueueUserWorkItem(o =>
                        {
                            try
                            {
                                Reader.Actions.Inventory.Stop();
                            }
                            catch (InvalidUsageException e)
                            {
                                e.PrintStackTrace();
                            }
                            catch (OperationFailureException e)
                            {
                                e.PrintStackTrace();
                            }
                        });
                    }

                }
            }
        }

    }
}
```

Migrating to a Combined RFD8500/RFD2000 RFID SDK

Introduction

The new RFID SDK library can be used with both RFD8500 and RFD2000 devices. This chapter provides the information necessary to update existing applications and develop new applications for use with the combined RFID SDK.

The RFID SDK supports the following devices.

- RFD8500 and the TC55/TC51 Android devices
- RFD2000 and the TC20 device.

Using Existing Applications With The RFID SDK

This new version of the RIFD SDK includes minimal changes that allow existing applications to integrate smoothly. No modifications are required for RFD2000 applications. Table 3 includes the modifications required for RFD8500 applications.

Table 3 Requirements for Integrating RFD8500 Applications

API	Change	Required Modification	Change Category
`GetAvailableRFIDReaderList`	Throws exception listing reader available.	Add try catch block around API. **Note**: This API should not be called from the main (UI) thread.	Minor
`Config.setAttribute`	Takes attribute value parameter as string.	Pass value as string. `setAttributeInfo.setAttvalue ("0")`	Minor

Access operations offset and count APIs:
`setByteCount`
`setByteOffset`
	Change word based API.	Use the following:
		`setCount`
		`setOffset`
			Minor

Migrating and Supporting RFD2000 Applications

See the tables below to understand the required changes for either migration or adding support for the first time.

NOTE: Current RFD8500 users should review the API differences below when porting applications from the RFD2000.

Table 4 SDK Instance Creation Overloading

Old SDK	New SDK	Notes
`new Readers()`	`new Readers()` `new Readers(this, ENUM_TRANSPORT.SERVICE_SERIAL);`	Bluetooth (BT) is the default `ENUM_TRANSPORT`. If the instance uses the old SDK code, the existing application can work as is without a change. First parameter: `'this'` is the Android application Context being passed and should be MainActivity context. Second parameter: `ENUM_TRANSPORT can be` 1. `BLUETOOTH` 2. `SERVICE_SERIAL` 3. `ALL` **Note**: The third `ENUM_TRANSPORT` value, **ALL**, is not useful for field scenarios as the application works with a specific type of reader. It is used for testing.

Table 5 Attribute API

Old SDK	New SDK	Notes
`SetAttribute` Accepts attribute value as integer.	`SetAttribute` Accepts attribute value as string.	Applications developed using the old RFD8500 SDK limited APIs to use integer type attributes only. The updated SDK allows setting any type of attribute.

Table 6 SDK instance disposal

Old SDK	New SDK	Notes
NA	`Dispose();`	Application should call this method when application exits or wants to release the SDK after disconnection with the reader. `readers.Dispose();`

Table 7 Model Specific APIs

RFD8500	RFD2000	Notes
`Config.setBeeperVolume` `Config.getBeeperVolume`	Not supported by reader.	set/get Beeper on RFD8500.
`Config.getBatchModeConfig` `Config.setBatchMode`	Not supported by reader.	set/get batch mode on RFD8500.
Supports barcode and RFID mode switching.	`Config.SetTriggerMode`	Trigger works for barcode or RFID tags.
Not supported by reader.	`Config.setLedBlinkEnable`	Blinks LED on TC20 on tag reads.

Summarizing Application Support for RFD8500 and RFD2000 Readers

NOTE: The following bullets assume that the application has prior knowledge of the type of reader model it is working with and acts accordingly.

- New applications support both the RFD8500 and RFD2000 readers.
- Applications developed for the RFD8500 now support the RFD2000.
  - The SDK is instantiated by calling Readers(). The new SDK creates a new instance of the SDK by specifying BT as the transport and the application receives the reader list via BT.
  - The application handles model specific API usage (see Table 7 above).
- Applications developed for the RFD2000 now support the RFD8500.
  - Applications are already using overloaded instance creation of SDK and disposal and can retrieve a BT paired device list by adding BT as the transport.
  - Applications handle model specific API usage (see Table 7 above).

Examples

The following example shows getting a reader list for both reader models based on the device. The application can create an instance of reader by checking the device type.

```
if (readers == null) {
    if (Build.DEVICE.contains("TC20"))
        readers = new Readers(this, ENUM_TRANSPORT.SERVICE_SERIAL);
    else
        readers = new Readers();
}
```

The following example shows model specific API inclusion and getting beeper volume for the RFD8500. The application can check the connected reader model name to exercise the supported API on a particular model to avoid conflict.

```
if(Application.mConnectedReader.ReaderCapabilities.getModelName().contains("RFD8500")) {
    Application.beeperVolume = Application.mConnectedReader.Config.getBeeperVolume();
    Application.batchMode =
Application.mConnectedReader.Config.getBatchModeConfig().getValue();
}
```