Getting Started with Zebra Bluetooth 123RFID Mobile iOS

FILENAME: Zebra_Bluetooth_123RFID_iOS_SDK_Getting_Started.doc

Version: 01.00.00

Date: 08-May-2023

Contents

1. Introduction

1.1 Purpose

This document aims to describe the configuration of XCode projects for the utilization of Zebra Bluetooth 123 RFID Mobile iOS SDK.

1.2 Scope

This document defines step-by-step instructions for setting up a new XCode project to work with Zebra Bluetooth 123 RFIDMobile iOS SDK.

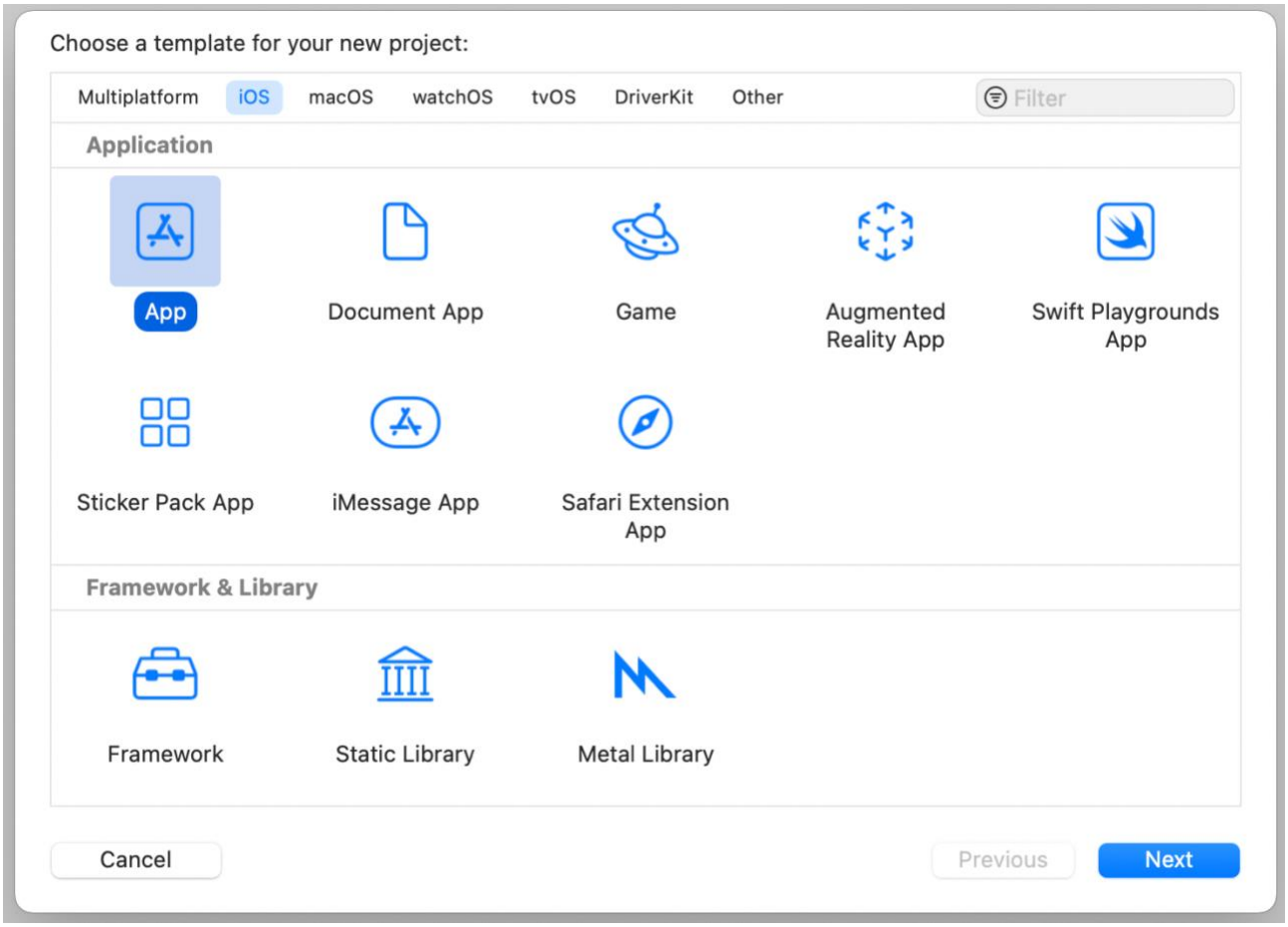
1.3 Acronyms, Abbreviations, and Definitions.

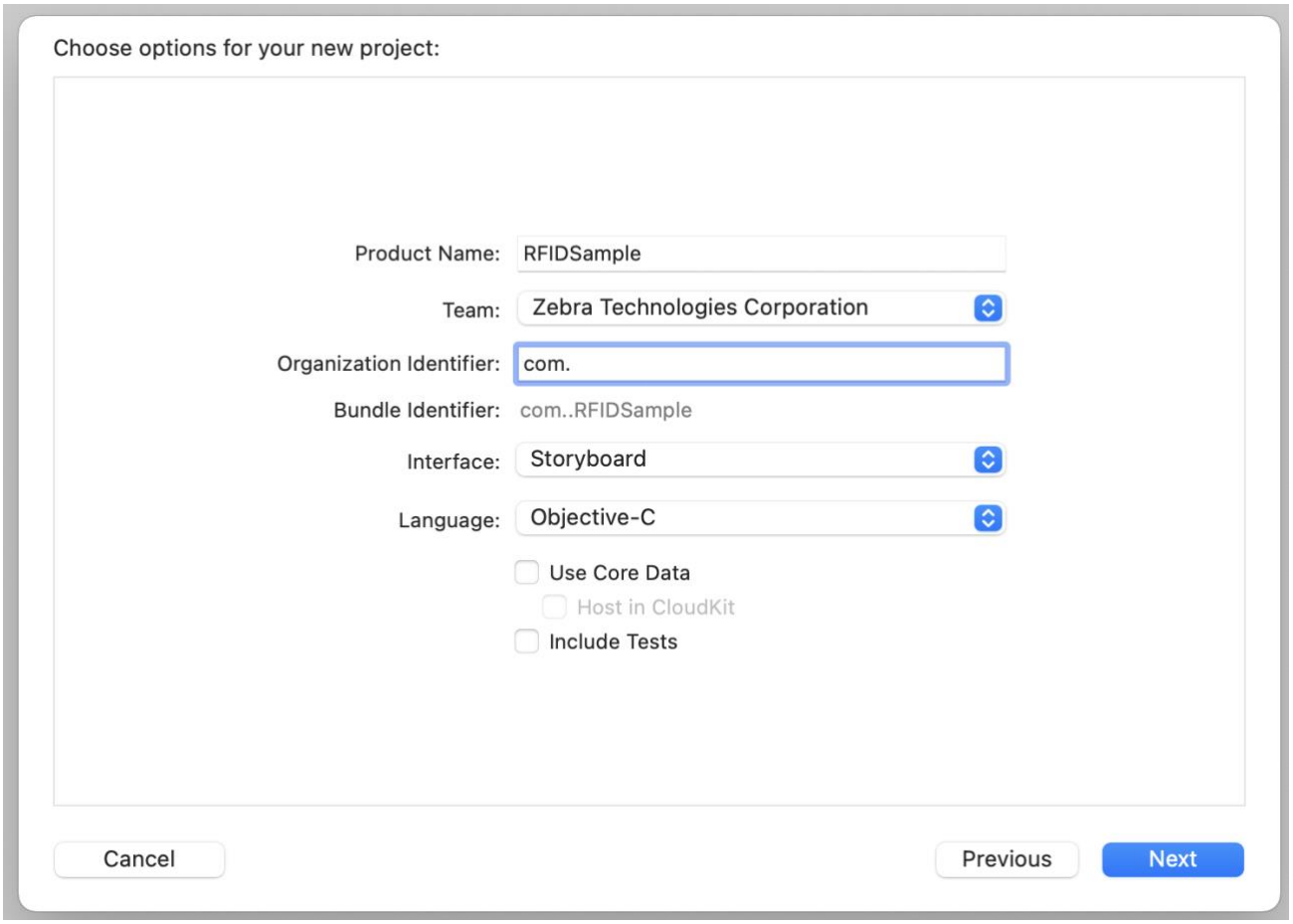
BT	Bluetooth
SDK	Software Development Kit

2. Setting up XCode project for SDK-based iOS application

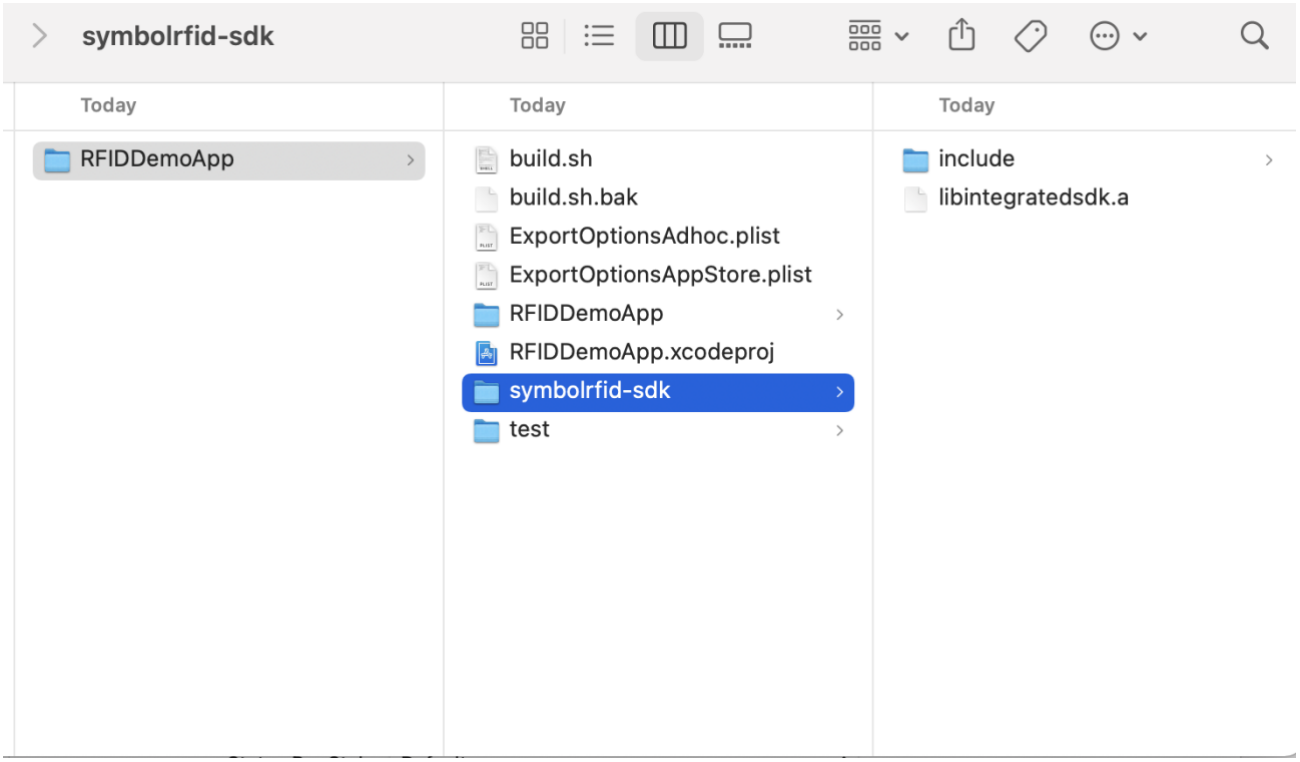
This section describes step-by-step instructions for setting up a new XCode project to work with Zebra Bluetooth 123 RFID iOS SDK.

2.1. Create new "iOS Application" project in XCode IDE

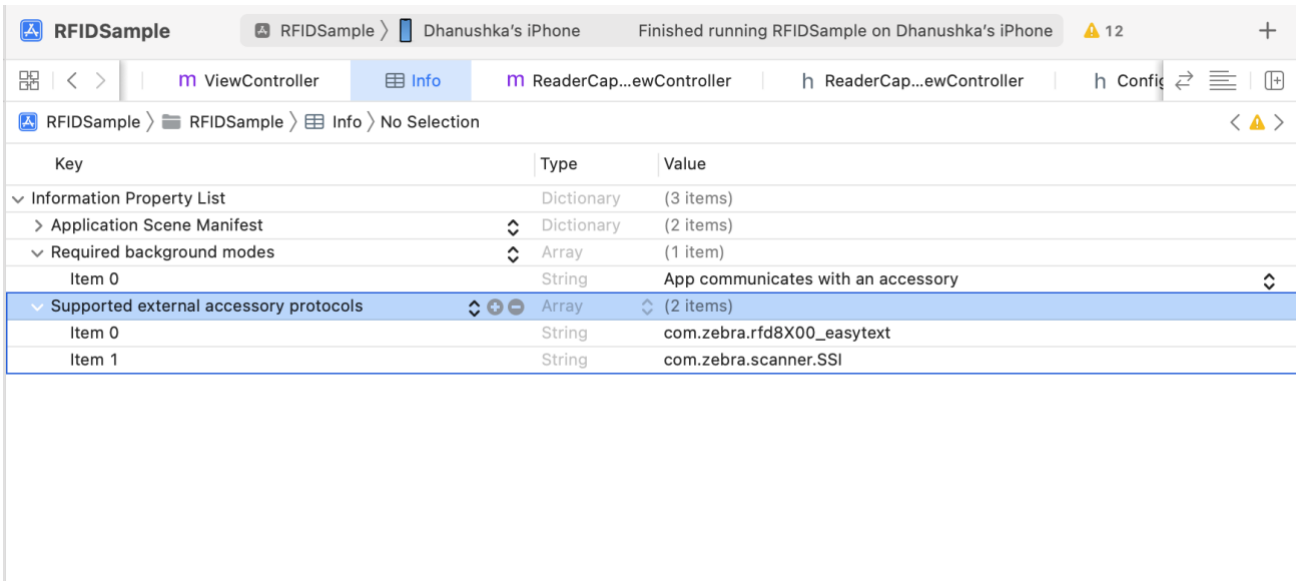




2.2. Copy `symbolrfid-sdk` folder with static library and headers from Zebra Bluetooth RFID iOS SDK installation directory to the root folder of your XCode project (note: symbolic link could also be used instead of copying)

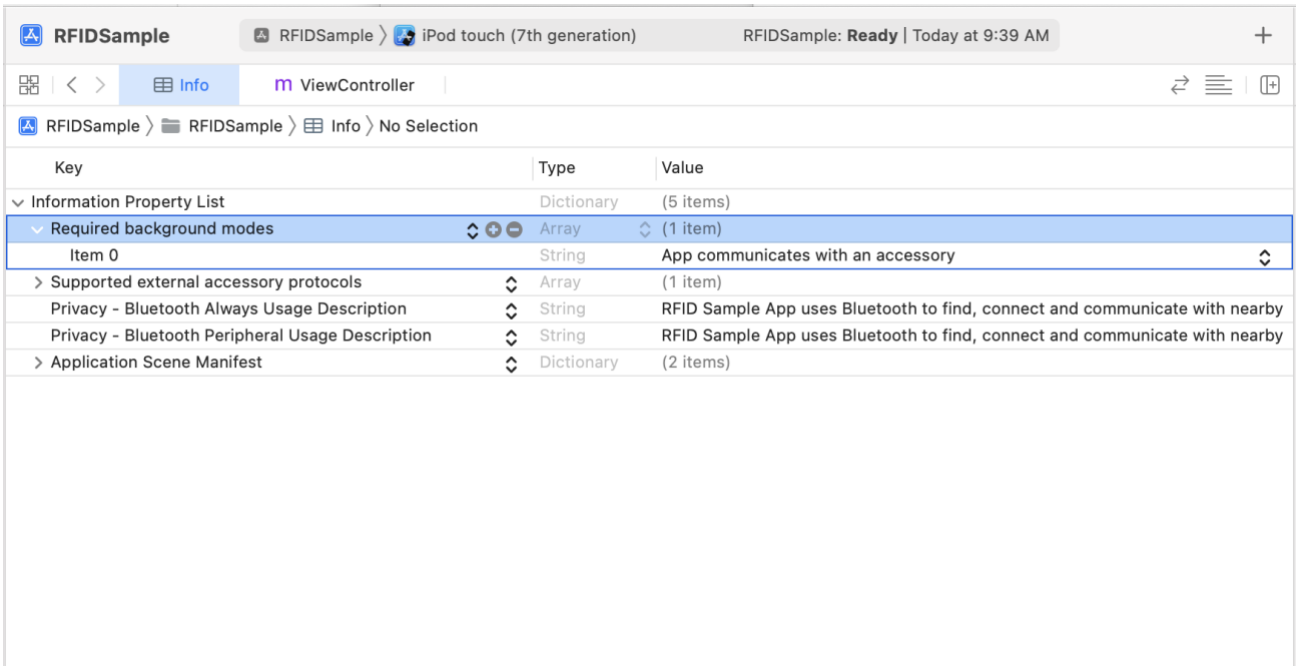


2.3. Configure your XCode project to support the "`com.zebra.rfd8X00_easytext and com.zebra.scanner.SSI`" external accessory communication protocol through including the `Supported external accessory protocols` key in your app's `Info.plist` file or via `[Info]` tab of your project settings



2.4. If your application is supposed to be able to communicate with BT RFID readers in a background mode configure your XCode project to declare the backround modes your app supports through

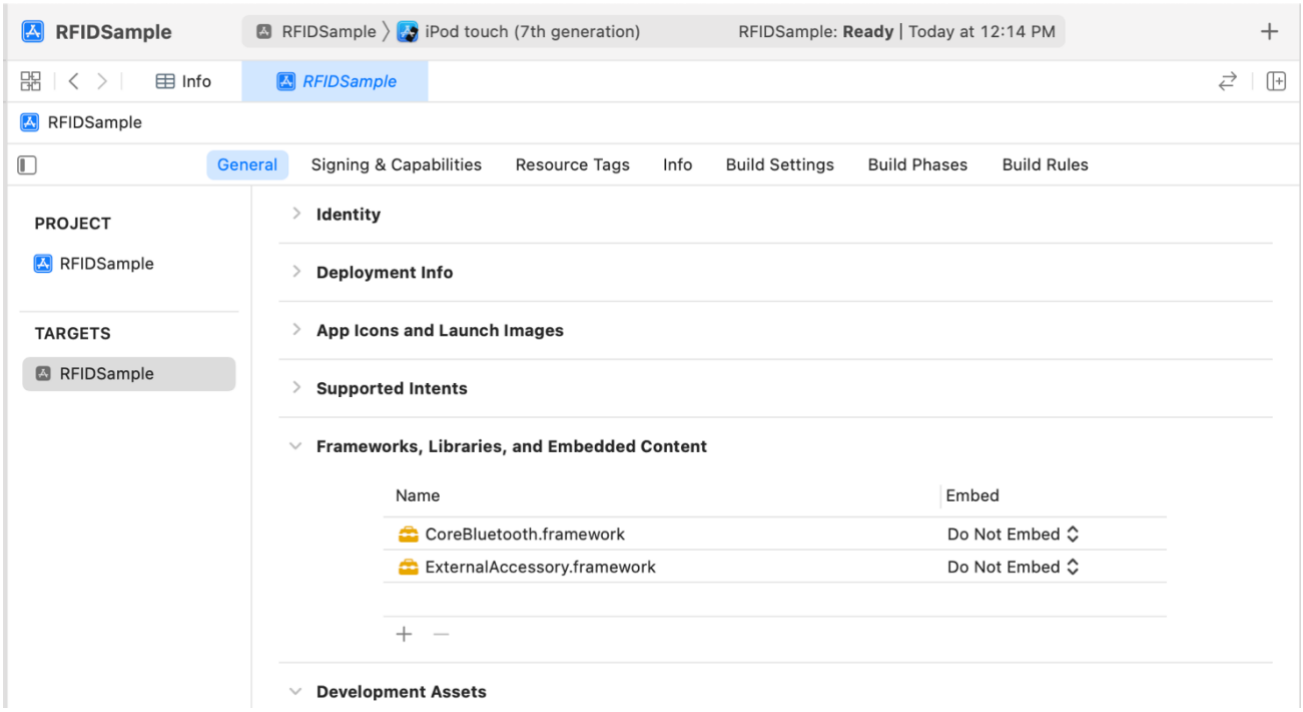
including the `Required background modes`key in your app's `Info.plist` file or via `[Info]` tab of your project settings



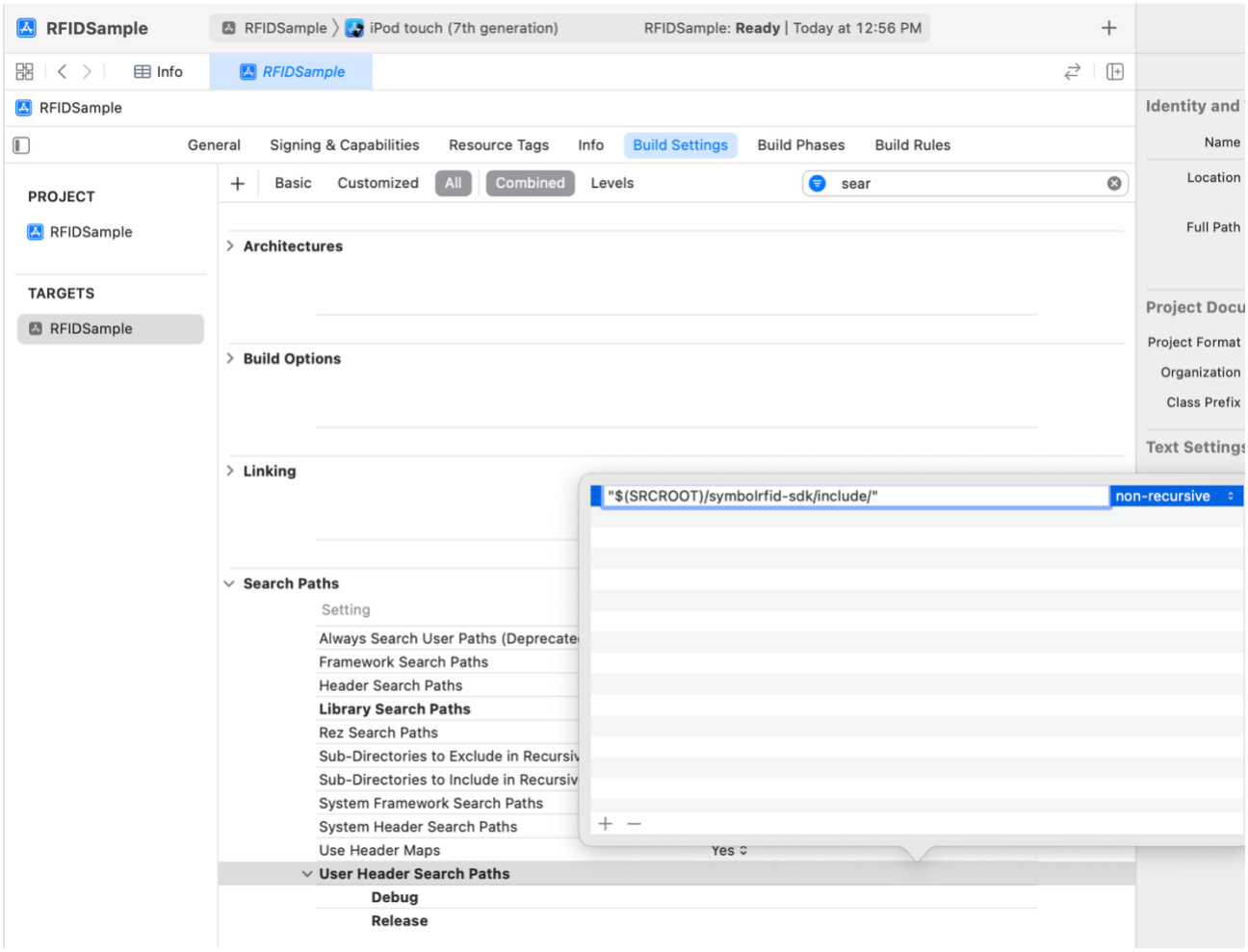
2.5. Configure your application to be linked with following default iOS frameworks that are required for utilization of Zebra Bluetooth RFID iOS SDK via `[Link Binary With Libraries]` section of `[Build Phases]` tab of your project settings:

- `ExternalAccessory.framework`

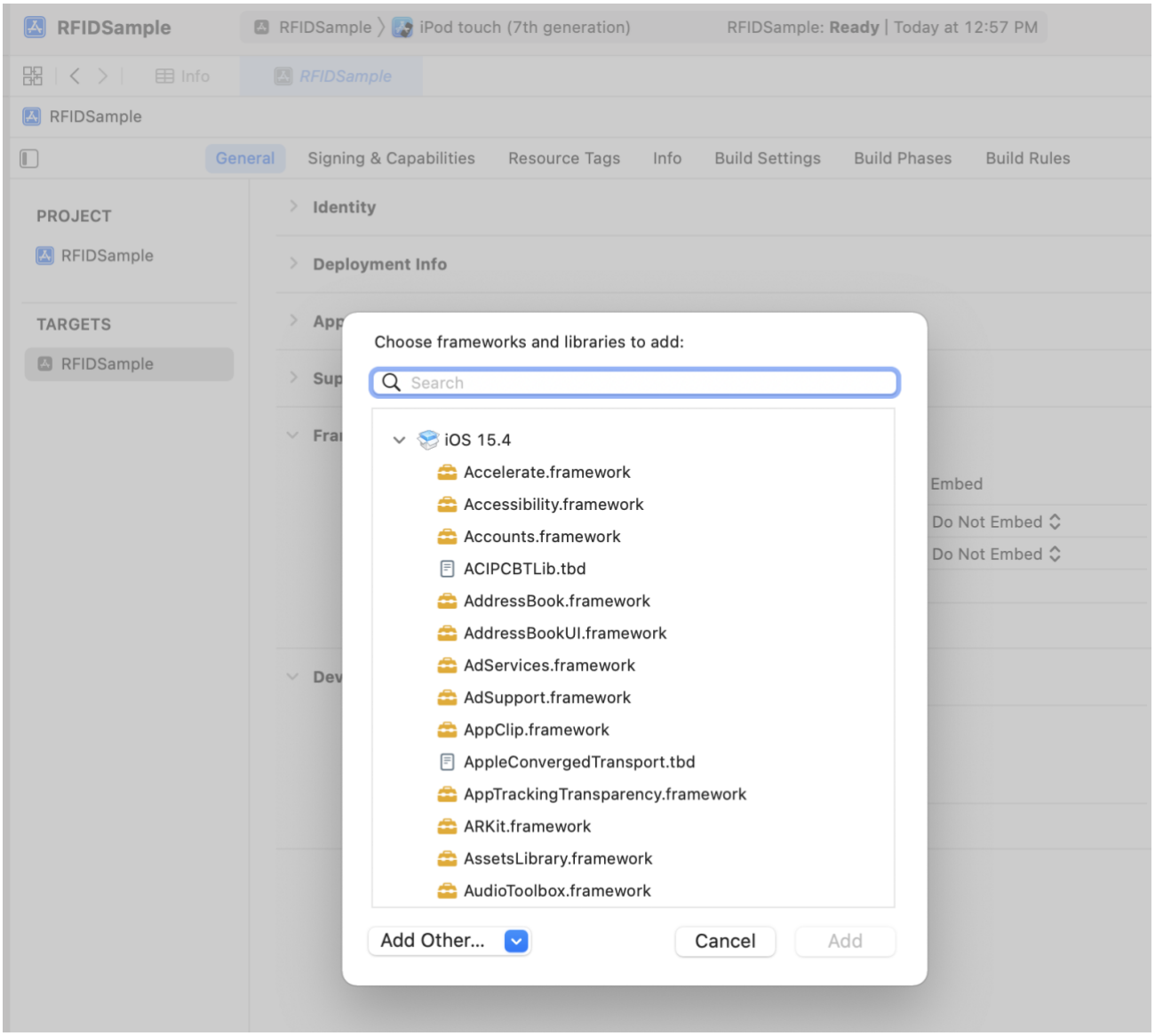
`-CoreBluetooth.framework`

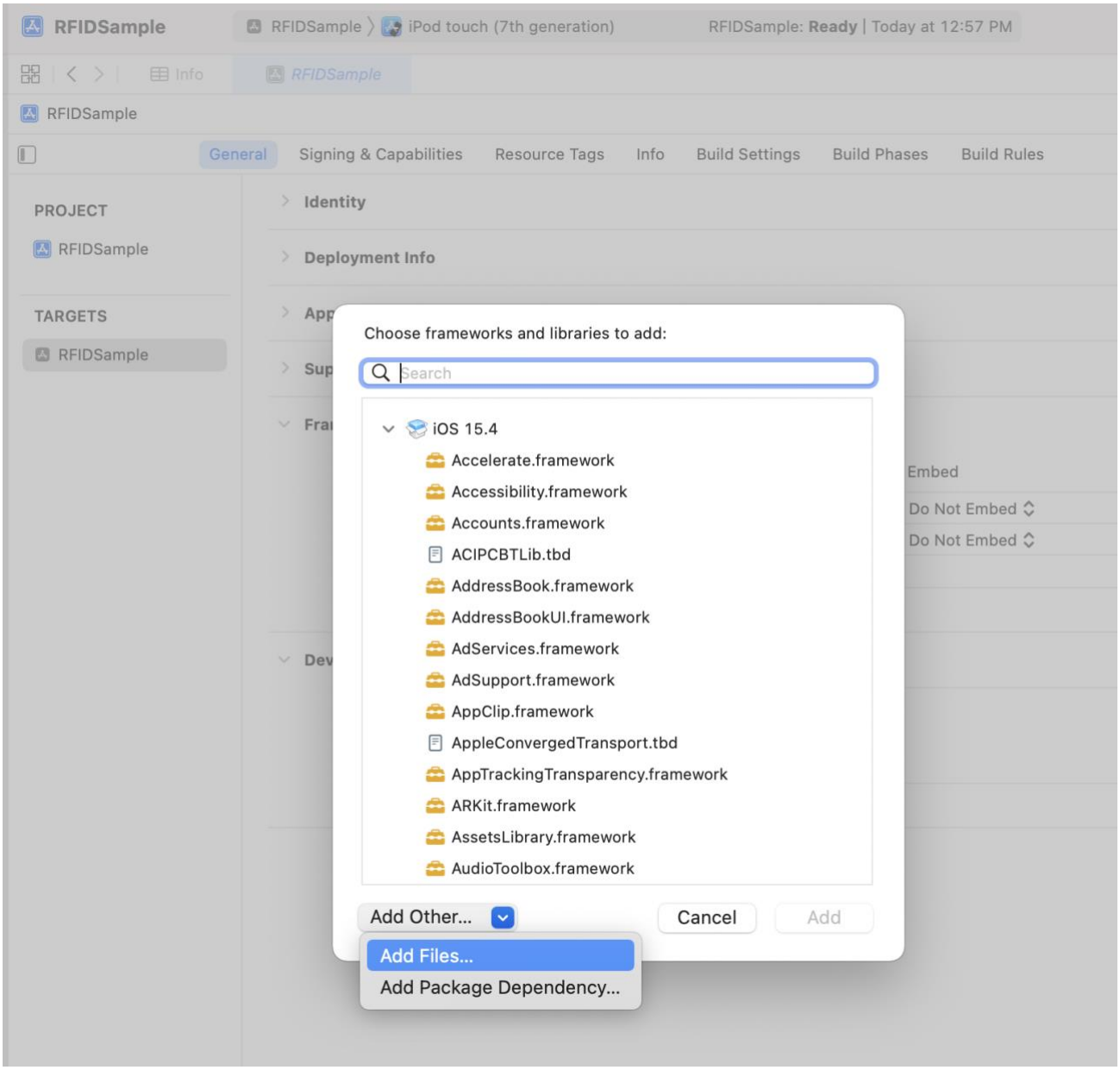


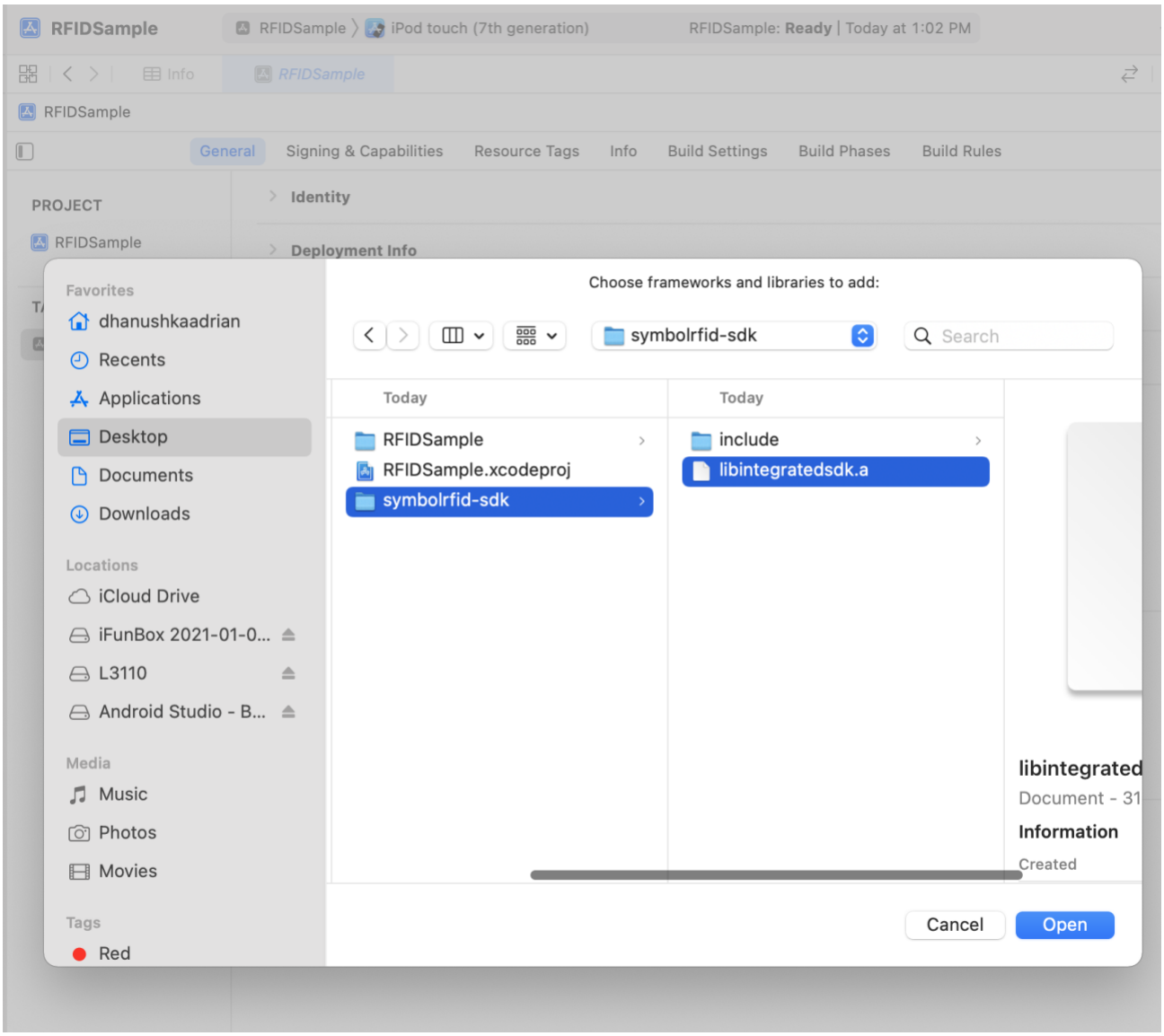
2.6. Configure your XCode project to make Zebra Bluetooth RFID iOS SDK headers available through `"$(SRCROOT)/symbolrfid-sdk/include/"` value of `[User Header Search Paths]` option in `[Search Paths]` section of `[Build Settings]` tab of your project settings

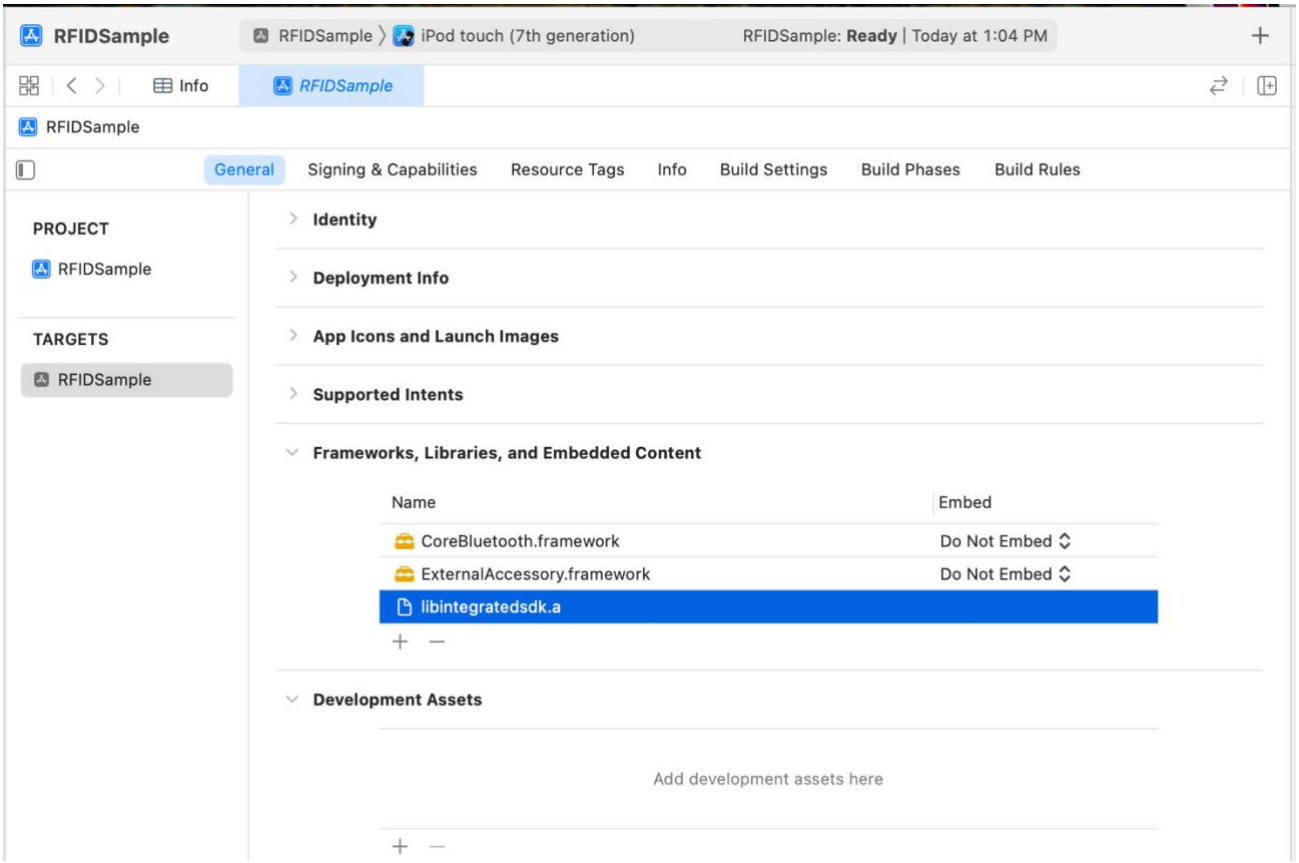


2.7. Configure your application to be linked with Zebra Bluetooth RFID iOS SDK static library through `[Link Binary With Libraries]` section of `[Build Phases]` tab of your project settings

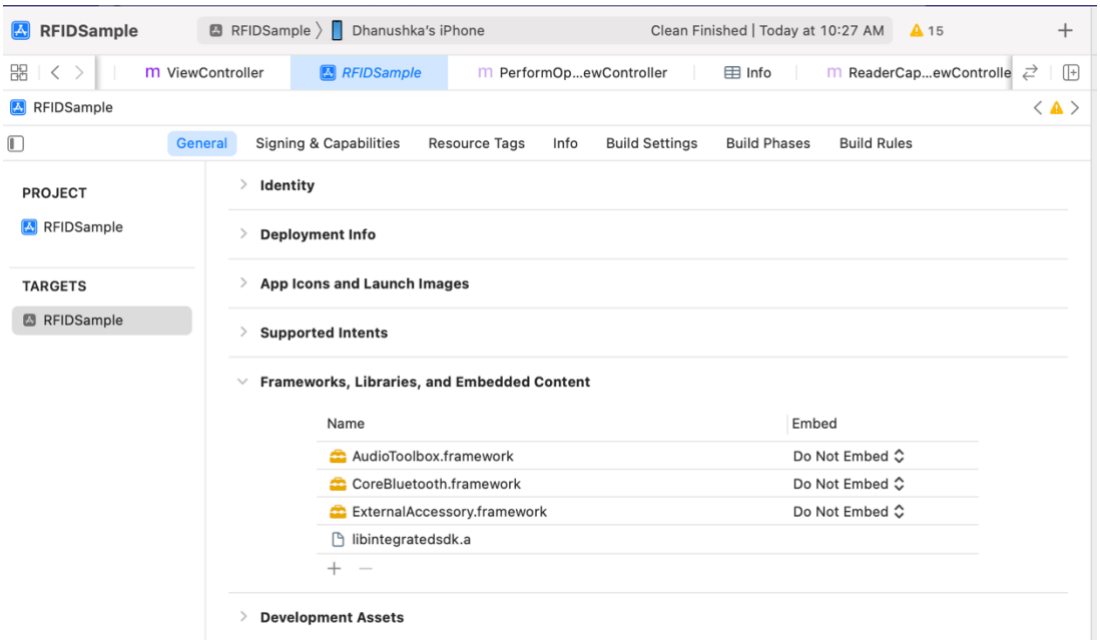


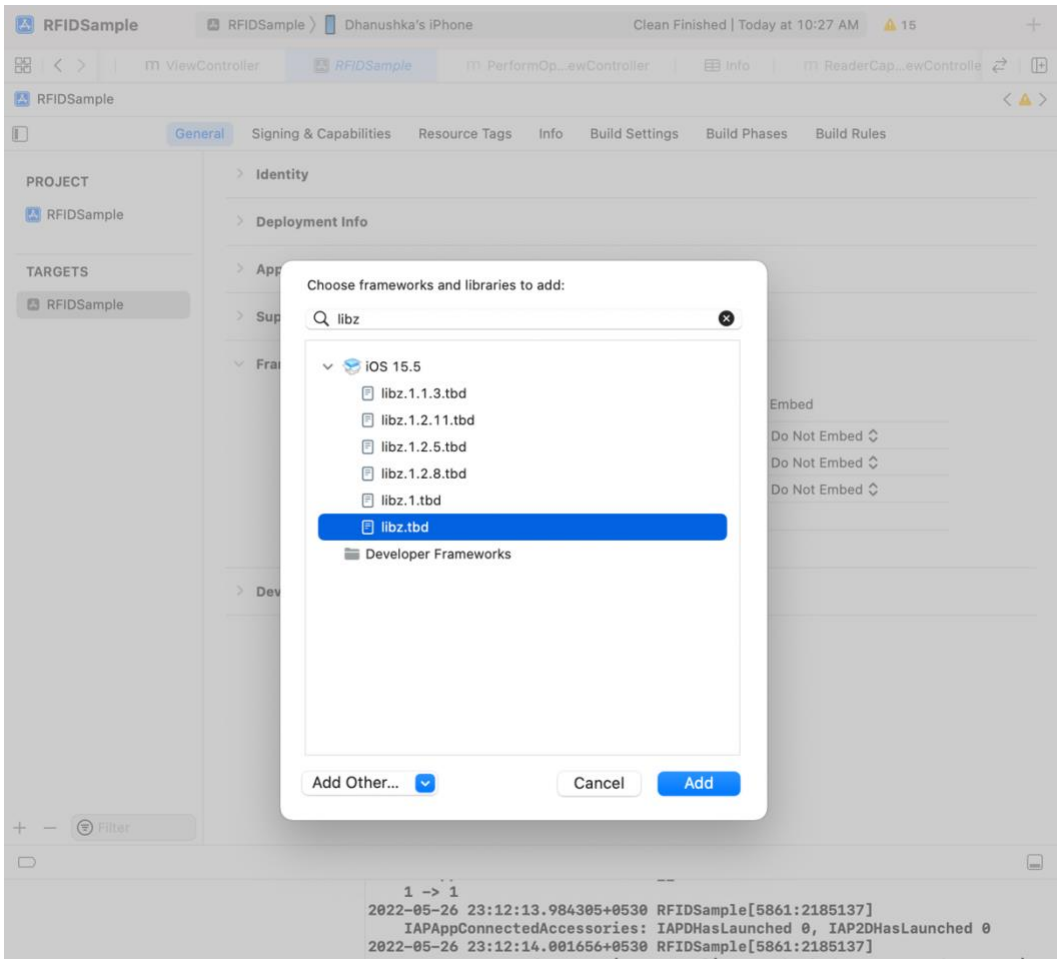






2.8. Add libz.tbd library.





3. RFID SDK API Calls

3.1 Implement *srfidISdkApiDelegateProtocol*

The SDK supports a set of asynchronous notifications to inform the application about RFID reader related events (e.g. reception of tag data, starting of radio operation etc) and connectivity related events (e.g. appearance of RFID reader). All supported callbacks are defined by *srfidISdkApiDelegate* Objective C protocol. In order to receive asynchronous notifications from the SDK the application shall perform following steps.

Step 1: create an object that implements *srfidISdkApiDelegateProtocol*

The ViewController.h class

```
#import <UIKit/UIKit.h>
#import "RfidSdkApiDelegate.h"

@interface ViewController : UIViewController<srfidISdkApiDelegate> {
}
@end
```

The Event ViewController.m class

```
#import "ViewController.h"
@implementation ViewController

- (void)srfidEventBatteryNotity:(int)readerID aBatteryEvent:(srfidBatteryEvent *)batteryEvent {
  // <#code#>
}

- (void)srfidEventCommunicationSessionEstablished:(srfidReaderInfo *)activeReader {
```

```
    NSLog(@"Reader Connected ");
}



- (void)srfidEventCommunicationSessionTerminated:(int)readerID {
    NSLog(@"Reader Disconnected ");
}

- (void)srfidEventMultiProximityNotify:(int)readerID aTagData:(srfidTagData *)tagData {
   // <#code#>
}

- (void)srfidEventProximityNotify:(int)readerID aProximityPercent:(int)proximityPercent {
   // <#code#>
}

- (void)srfidEventReadNotify:(int)readerID aTagData:(srfidTagData *)tagData {
   // <#code#>
}

- (void)srfidEventReaderAppeared:(srfidReaderInfo *)availableReader {
   // <#code#>
}

- (void)srfidEventReaderDisappeared:(int)readerID {
   // <#code#>
}

- (void)srfidEventStatusNotify:(int)readerID aEvent:(SRFID_EVENT_STATUS)event
aNotification:(id)notificationData {
    //<#code#>
}
```

```
- (void)srfidEventTriggerNotify:(int)readerID aTriggerEvent:(SRFID_TRIGGEREVENT)triggerEvent {
    //<#code#>
}

@end
```

Step 2: register the created object as notification receiver via *srfidSetDelegate* API function.

```
-(void)registerOfcallbackInterfaceWithSDK {

    /* registration of callback interface with SDK */
    [apiInstance srfidSetDelegate:self];
}
```

Step 3: subscribe for asynchronous event of particular types via *srfidSubscribeForEvents* API function.

If a particular object is registered as a notification receiver the SDK will call the corresponding method of the registered object when a particular event occurs if the application is subscribed for events of this type. The SDK may deliver asynchronous events on a main thread or on one of SDK helper threads so the object that implements *srfidISdkApiDelegate* protocol shall be thread-safe.

```
-(void)subcribeForEvent {

    int notifications_mask_reader_connection = SRFID_EVENT_READER_APPEARANCE |
```

```
                    SRFID_EVENT_READER_DISAPPEARANCE |
                    SRFID_EVENT_SESSION_ESTABLISHMENT |
                    SRFID_EVENT_SESSION_TERMINATION;
    [apiInstance srfidSubsribeForEvents:notifications_mask_reader_connection];
    /* subscribe for battery and handheld trigger related events */
    [apiInstance srfidSubsribeForEvents:(SRFID_EVENT_MASK_BATTERY |
SRFID_EVENT_MASK_TRIGGER)];
    [apiInstance srfidSubsribeForEvents:(SRFID_EVENT_MASK_READ | SRFID_EVENT_MASK_STATUS |
SRFID_EVENT_MASK_STATUS_OPERENDSUMMARY)];
    [apiInstance srfidSubsribeForEvents:(SRFID_EVENT_MASK_TEMPERATURE |
SRFID_EVENT_MASK_POWER | SRFID_EVENT_MASK_DATABASE)];
    [apiInstance srfidSubsribeForEvents:(SRFID_EVENT_MASK_PROXIMITY)];
    [apiInstance srfidSubsribeForEvents:(SRFID_EVENT_MASK_TRIGGER)];
    [apiInstance srfidSubsribeForEvents:(SRFID_EVENT_MASK_BATTERY)];
    [apiInstance srfidSubsribeForEvents:(SRFID_EVENT_MASK_MULTI_PROXIMITY)];


}
```

4. Connectivity Management

4.1 Set operation mode

Zebra Bluetooth RFID iOS SDK is designed to support interaction with RFID readers operating in either BT MFi or BT LE mode. The SDK shall be intentionally configured to enable communication with a particular type of RFID readers via *srfidSetOperationalMode* API function. If operating mode of the SDK is not configured the SDK will remain disabled and will not be able to communicate with RFID readers in neither BT MFi nor BT LE modes.

Following example demonstrates enabling interaction with RFID readers in BT MFi mode.

```
[apiInstance srfidSetOperationalMode:SRFID_OPMODE_MFI];
```

4.2 Get available readers

Following terms are introduced to distinguish RFID readers that are seen by the SDK via OS API and RFID readers with that the SDK has established a logical communication session and thus is able to interact. A RFID reader is called available if it is already connected to the iOS device via Bluetooth. Such RFID reader is seen by the SDK and the SDK can establish a logical communication session to interact with the RFID reader. If a logical communication session is established with already connected via Bluetooth RFID reader the RFID reader is called active.

The SDK supports simultaneous interaction with multiple active RFID readers. To distinguish various RFID readers the SDK assigns the unique integer identifier for each RFID reader when it becomes available first time.

The SDK maintains internal lists of active and available RFID readers. Following example demonstrates reception of lists of active and available RFID readers from the SDK.

```
-(void)getAvialableReaderList{


    /* allocate an array for storage of list of available RFID readers */
    NSMutableArray *available_readers = [[NSMutableArray alloc] init];
    /* allocate an array for storage of list of active RFID readers */
    NSMutableArray *active_readers = [[NSMutableArray alloc] init];

    /* retrieve a list of available readers */
    [apiInstance srfidGetAvailableReadersList:&available_readers];
    /* retrieve a list of active readers */
    [apiInstance srfidGetActiveReadersList:&active_readers];

    /* merge active and available readers to a single list */
    NSMutableArray *readers = [[NSMutableArray alloc] init];
    [readers addObjectsFromArray:active_readers];
    [readers addObjectsFromArray:available_readers];
    for (srfidReaderInfo *info in readers) {
        /* print the information about RFID reader represented by srfidReaderInfo object */
        NSLog(@"RFID reader is %@: ID = %d name = %@\n", (([info isActive] == YES) ? @"active" :
@"available"), [info getReaderID], [info getReaderName]);

        lable_reader_list.text = [info getReaderName];
        readerId = [info getReaderID];
    }
```

```
}
```

4.3 Enable available readers detection

The SDK supports automatic detection of appearance and disappearance of available RFID readers. When "Available readers detection" option is enabled the SDK will update its internal list of available RFID readers and deliver a corresponding asynchronous notification once it detects connection or disconnection of a particular RFID reader to the iOS device via Bluetooth. If the option is disabled the SDK updates its internal list of available RFID readers only when it is requested by an application via *srfidGetAvailableReadersList* API function. Following example demonstrates enabling of automatic detection and processing of corresponding asynchronous notifications.

```
[apiInstance srfidEnableAvailableReadersDetection:YES];
```

4.4 Enable automatic communication session reestablishment

The SDK supports "Automatic communication session reestablishment" option. When the option is enabled the SDK will automatically establish a logical communication session with the last active RFID reader that had unexpectedly disappeared once the RFID reader will be recognized as available. If "Available readers detection" option is enabled the RFID reader will be recognized as available

automatically when it becomes connected via Bluetooth. Otherwise the SDK will add the RFID reader to the list of available RFID readers only during discovery procedure requested by the application via *srfidGetAvailableReadersList* API. The option has no effect if the application has intentionally terminate a communication session with the active RFID reader via *srfidTerminateCommunicationSession* API function. The "Automatic communication session reestablishment" option is configured via *srfidEnableAutomaticSessionReestablishment* API function.

```
[apiInstance srfidEnableAutomaticSessionReestablishment:YES];
```

5. Knowing the Reader related Information

5.1 Knowing the Software Version

The SDK provides an ability to retrieve information about software versions of various components of a particular active RFID reader. Software version related information could be retrieved via *srfidGetReaderVersionInfo* API function as demonstrated in the following example.

```
-(void)getReaderInformation {
    /* identifier of one of active RFID readers is supposed to be stored in m_ReaderId variable */

    /* allocate object for storage of version related information */
    srfidReaderVersionInfo *version_info = [[srfidReaderVersionInfo alloc] init];
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;

    /* retrieve version related information */
    SRFID_RESULT result = [_apiInstance srfidGetReaderVersionInfo:_connectedRederID
aReaderVersionInfo:&version_info aStatusMessage:&error_response];

    if ((result != SRFID_RESULT_RESPONSE_TIMEOUT) && (result != SRFID_RESULT_FAILURE))
    {
        NSLog(@"Time out or Failure");
    }
    if (SRFID_RESULT_SUCCESS == result) {
        /* print the received version related information */
        NSLog(@"Device version: %@\n", [version_info getDeviceVersion]);
        NSLog(@"NGE version: %@\n", [version_info getNGEVersion]);
        NSLog(@"Bluetooth version: %@\n", [version_info getBluetoothVersion]);
        textView_reader_information.text = [NSString stringWithFormat:@"Firmware version: %@\n NGE
version: %@\n Bluetooth version: %@\n", [version_info getDeviceVersion], [version_info getNGEVersion],
[version_info getBluetoothVersion]];
    }
```

```
    if (SRFID_RESULT_READER_NOT_AVAILABLE == result) {


        NSLog(@"RFID reader is not available\n");
        textView_reader_information.text = @"RFID readernot available";
    }



}
```

5.2 Knowing the Reader Capabilities

The SDK provides an ability to retrieve the capabilities (or read-only properties) of a particular active RFID reader. The reader capabilities include the following:

- Serial number
- Model name
- Manufacturer
- Manufacturing date
- Device name
- ASCII protocol version
- Number of select records (pre-filters)
- Minimal and maximal antenna power levels (in 0.1 dbm units)
- Step for configuration of antenna power level (in 0.1 dbm units)
- Version of air protocol
- Bluetooth address
- Maximal number of operations to be combined in a sequence.

The reader capabilities could be retrieved via *srfidGetReaderCapabilitiesInfo* API function as demonstrated in the following example.

```
-(void)getReaderCapabilities {


    /* allocate object for storage of capabilities information */
    srfidReaderCapabilitiesInfo *capabilities = [[srfidReaderCapabilitiesInfo alloc] init];


    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
```

```
    /* retrieve capabilities information */
    SRFID_RESULT srfid_result = SRFID_RESULT_FAILURE;
    srfid_result = [_apiInstance srfidGetReaderCapabilitiesInfo:_connectedRederID
aReaderCapabilitiesInfo:&capabilities aStatusMessage:&error_response];

    if ((srfid_result != SRFID_RESULT_RESPONSE_TIMEOUT) && (srfid_result !=
SRFID_RESULT_FAILURE))
    {
        NSLog(@"Time out or Failure");
    }

    if (srfid_result == SRFID_RESULT_SUCCESS)
    {
        NSLog(@"Serial number: %@\n", [capabilities getSerialNumber]);
        NSLog(@"Model: %@\n", [capabilities getModel]);
        NSLog(@"Manufacturer: %@\n", [capabilities getManufacturer]);
        NSLog(@"Manufacturing date: %@\n", [capabilities getManufacturingDate]);
        NSLog(@"Scanner name: %@\n", [capabilities getScannerName]);
        NSLog(@"Ascii version: %@\n", [capabilities getAsciiVersion]);
        NSLog(@"Air version: %@\n", [capabilities getAirProtocolVersion]);
        NSLog(@"Bluetooth address: %@\n", [capabilities getBDAddress]);
        NSLog(@"Select filters number: %d\n", [capabilities getSelectFilterNum]);
        NSLog(@"Max access sequence: %d\n", [capabilities getMaxAccessSequence]);
        NSLog(@"Power level: min = %d; max = %d; step = %d\n", [capabilities getMinPower], [capabilities
getMaxPower], [capabilities getPowerStep]);

        textView_reader_capabilities.text = [NSString stringWithFormat:@"Serial number: %@\n Model: %@\n
Bluetooth address: %@\n",[capabilities getSerialNumber],[capabilities getModel],[capabilities
getBDAddress]];
    }

}
```

5.3 Knowing Supported Regions

The RFID reader could be configured to operate in a various countries. The SDK provides an ability to retrieve the list of regions supported by a particular active RFID reader. The list of supported regions could be retrieved via *srfidGetSupportedRegions* API function as demonstrated in the following example.

```
-(void)getSupportRegion {
    /* allocate object for storage of region information */
    NSMutableArray *regions = [[NSMutableArray alloc] init];

    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;

    /* retrieve supported regions */
    SRFID_RESULT result = [_apiInstance srfidGetSupportedRegions:_connectedRederID
aSupportedRegions:&regions aStatusMessage:&error_response];

    if (SRFID_RESULT_SUCCESS == result) {
        /* print supported regions information */
        NSLog(@"Number of supported regions: %lu\n", (unsigned long)[regions count]);
        for (srfidRegionInfo *info in regions)
        {
            NSLog(@"Regions [%@] is supported: %@\n", [info getRegionName],  [info getRegionCode]);

        }
        NSString * result = [[regionsDetatilsArray valueForKey:@"description"]
componentsJoinedByString:@"\n"];

    }
    else if (SRFID_RESULT_RESPONSE_ERROR == result) {
        NSLog(@"Error response from RFID reader: %@\n", error_response);
    }
    else if (SRFID_RESULT_RESPONSE_TIMEOUT == result) {
        NSLog(@"Timeout occurs during communication with RFID reader\n");
    }
```

```
    else if (SRFID_RESULT_READER_NOT_AVAILABLE == result) {
        NSLog(@"RFID reader with id = %d is not available\n", _connectedRederID);
    }
    else {
        NSLog(@"Request failed\n");
    }

}
```

As the RFID reader could be configured to operate on a particular radio channels in some of countries the SDK provides an ability to retrieve the detailed information regarding one of regions supported by a particular active RFID reader. The detailed information includes a set of channel supported in the region and allowance of hopping configuration. This information could be retrieved via *srfidGetRegionInfo* API function as demonstrated in the following example.

```
-(void)getSupportChannelListForGivenRegion {

    /* allocate object for storage of supported channels information */
    NSMutableArray *channels = [[NSMutableArray alloc] init];
    BOOL hopping = NO;
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
    /* retrieve detailed information about region specified by "USA" region code */
    SRFID_RESULT result = [_apiInstance srfidGetRegionInfo:_connectedRederID aRegionCode:@"AUS"
aSupportedChannels:&channels aHoppingConfigurable:&hopping aStatusMessage:&error_response];

    if (SRFID_RESULT_SUCCESS == result) {
        /* print retrieved detailed information */
        NSLog(@"Hopping configuration is: %@\n", ((YES == hopping) ? @"supported" : @"NOT supported"));

        for (NSString *str_channel in channels)
        {
            NSLog(@"Supported channel: %@\n", str_channel);
        }
        NSString * result = [[channels valueForKey:@"description"] componentsJoinedByString:@"\n"];
```

```
        textView_reader_support_channel.text = result;
    }
    else if (SRFID_RESULT_RESPONSE_ERROR == result) {
        NSLog(@"Error response from RFID reader: %@\n", error_response);
    }
    else if (SRFID_RESULT_RESPONSE_TIMEOUT == result) {
        NSLog(@"Timeout occurs during communication with RFID reader\n");
    }
    else if (SRFID_RESULT_READER_NOT_AVAILABLE == result) {
        NSLog(@"RFID reader with id = %d is not available\n", _connectedRederID);
    }
    else {
        NSLog(@"Request failed\n");
    }
}
```

5.4 Knowing Supported Link Profiles

An antenna of the RFID reader could be configured to operate in various RF modes (link profiles). The SDK provides an ability to retrieved the list of link profiles (RF modes) supported by a particular active RFID reader. The list of supported link profiles could be retrieved via *srfidGetSupportedLinkProfiles* API function as demonstrated in the following example.

```
-(void)getSupportLinkProfile {


    /* allocate object for storage of link profiles information */
    NSMutableArray *profiles = [[NSMutableArray alloc] init];
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
    /* retrieve supported link profiles */
    SRFID_RESULT result = [_apiInstance srfidGetSupportedLinkProfiles:_connectedRederID
aLinkProfilesList:&profiles aStatusMessage:&error_response];


    if (SRFID_RESULT_SUCCESS == result) {
        /* print retrieved information about supported link profiles */
        NSLog(@"Number of supported link profiles: %lu\n", (unsigned long)[profiles count]);
```

```
        for (srfidLinkProfile *profile_info in profiles) {
            NSLog(@"RF mode index: %d\n", [profile_info getRFModeIndex]);
            NSLog(@"BDR: %d\n", [profile_info getBDR]);
            NSLog(@"PIE: %d\n", [profile_info getPIE]);
            NSLog(@"Tari: min = %d; max = %d; step = %d\n", [profile_info getMinTari], [profile_info getMaxTari],
[profile_info getStepTari]);
            NSLog(@"EPCHAGT&CConformance: %@\n", ((NO == [profile_info getEPCHAGTCConformance]) ?
@"NO" : @"YES"));
            NSLog(@"Divide Ratio: %@\n", [profile_info getDivideRatioString]);
            NSLog(@"FLM: %@\n", [profile_info getForwardLinkModulationString]);
            NSLog(@"M: %@\n", [profile_info getModulationString]);
            NSLog(@"Spectral Mask indicator: %@\n", [profile_info getSpectralMaskIndicatorString]);
        }
    }
    else if (SRFID_RESULT_RESPONSE_ERROR == result) {
        NSLog(@"Error response from RFID reader: %@\n", error_response);
    }
    else if (SRFID_RESULT_RESPONSE_TIMEOUT == result) {
        NSLog(@"Timeout occurs during communication with RFID reader\n");
    }
    else if (SRFID_RESULT_READER_NOT_AVAILABLE == result) {
        NSLog(@"RFID reader with id = %d is not available\n", _connectedRederID);
    }
    else {
        NSLog(@"Request failed\n");
    }

}
```

5.5 Knowing Battery Status

A particular active RFID reader could send an asynchronous notification regarding battery status. The SDK will inform the application about received asynchronous battery status event if the application has subscribed for events of this type. The SDK also provides an ability to cause a particular active RFID

reader to immediately send information about current battery status. The following example demonstrates both requesting and processing of asynchronous battery status related notifications.

```
-(void)getBatteryStatus {

    /* subscribe for battery related events */
    [apiInstance srfidSubsribeForEvents:SRFID_EVENT_MASK_BATTERY];
    /* cause RFID reader to generate asynchronous battery status notification */
    SRFID_RESULT result = [apiInstance srfidRequestBatteryStatus:connectedReaderId];

    if (SRFID_RESULT_SUCCESS == result) {
        NSLog(@"Request succeed\n");
    }
    else {
        NSLog(@"Request failed\n");
    }

}

- (void)srfidEventBatteryNotity:(int)readerID aBatteryEvent:(srfidBatteryEvent
*)batteryEvent {

    /* print the received information regarding battery status */
    NSLog(@"Battery status event received from RFID reader with ID = %d\n", readerID);
    NSLog(@"Battery level: %d\n", [batteryEvent getPowerLevel]);
    NSLog(@"Charging: %@\n", ((NO == [batteryEvent getIsCharging]) ? @"NO" :
@"YES"));
    NSLog(@"Event cause: %@\n", [batteryEvent getEventCause]);
    dispatch_async(dispatch_get_main_queue(),^{
        self->textView_reader_battery_status.text = [NSString stringWithFormat:@"Battery
level: %d ", [batteryEvent getPowerLevel]];
```

```
    });

}
```

6. Configuring the Reader

Zebra Bluetooth RFID iOS SDK API supports managing of various RFID reader parameters including:

- Antenna parameters
- Singulation parameters
- Start and stop triggers parameters
- Tag report parameters
- Regulatory parameters
- Pre-filters
- Beeper.

6.1 Antenna Configuration

Following antenna related settings could be configured via the SDK:

- Output power level (in 0.1 dbm units)
- Index of selected link profile (RF mode)
- Application of pre-filters (select records)
- Tari (Type-A reference interval).

Tari value shall be set in accordance with the selected link profile, i.e. tari value shall be in the interval between minimal and maximal tari values specified by the selected link profile. If step size is supported by the selected link profile, the tari value must be a multiple of step size. Antenna settings could be retrieved and set via *srfidGetAntennaConfiguration* and *srfidSetAntennaConfiguration* API function accordingly.

Following example demonstrates retrieving current antenna settings and setting of antenna configuration with minimal output power and one of supported link profiles.

```
-(void)getAntennaConfig {

    /* allocate object for storage of antenna settings */

    srfidAntennaConfiguration *antenna_cfg = [[srfidAntennaConfiguration alloc] init];


    /* an object for storage of error response received from RFID reader */

    NSString *error_response = nil;


    /* retrieve antenna configuration */

    SRFID_RESULT result = [_apiInstance srfidGetAntennaConfiguration:_connectedRederID
aAntennaConfiguration:&antenna_cfg aStatusMessage:&error_response];
```

```
        if (SRFID_RESULT_SUCCESS == result) {
            /* antenna configuration received */
            NSLog(@"Antenna power level: %1.1f\n", [antenna_cfg getPower]/10.0);
            NSLog(@"Antenna RF mode index: %d\n", [antenna_cfg getLinkProfileIdx]);
            NSLog(@"Antenna tari: %d\n", [antenna_cfg getTari]);
            NSLog(@"Antenna pre-filters application: %@", ((NO == [antenna_cfg getDoSelect]) ? @"NO" :
@"YES"));
        }
        else if (SRFID_RESULT_RESPONSE_ERROR == result) {
            NSLog(@"Error response from RFID reader: %@\n", error_response);
        }
        else if (SRFID_RESULT_RESPONSE_TIMEOUT == result) {
            NSLog(@"Timeout occurs during communication with RFID reader\n");
        }
        else if (SRFID_RESULT_READER_NOT_AVAILABLE == result) {
            NSLog(@"RFID reader with id = %d is not available\n", _connectedRederID);
        }
        else {
            NSLog(@"Request failed\n");
        }
}


-(void)setAntennaConfig {
    /* allocate object for storage of antenna settings */
    srfidAntennaConfiguration *antenna_cfg = [[srfidAntennaConfiguration alloc] init];
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
    /* RF mode index to be set */
    int link_profile_idx = 0;
    /* tari to be set */
    int tari = 0;
    /* 20.0 dbm power level to be set */
```

```
int power = 200;

/* allocate object for storage of link profiles information */
NSMutableArray *profiles = [[NSMutableArray alloc] init];
/* retrieve supported link profiles */
SRFID_RESULT result  = [_apiInstance srfidGetSupportedLinkProfiles:_connectedRederID
aLinkProfilesList:&profiles aStatusMessage:&error_response];

if (SRFID_RESULT_SUCCESS == result) {
   if (0 < [profiles count]) {
      srfidLinkProfile *profile = (srfidLinkProfile*)[profiles lastObject];
      link_profile_idx = [profile getRFModeIndex];
      tari = [profile getMaxTari];
      }
}

/* allocate object for storage of capabilities information */
srfidReaderCapabilitiesInfo *capabilities = [[srfidReaderCapabilitiesInfo alloc] init];

/* retrieve capabilities information */
result = [_apiInstance srfidGetReaderCapabilitiesInfo:_connectedRederID
aReaderCapabilitiesInfo:&capabilities aStatusMessage:&error_response];
if (SRFID_RESULT_SUCCESS == result) {
   power = [capabilities getMinPower];
}
/* prepare an object with desired antenna parameters */
antenna_cfg = [[srfidAntennaConfiguration alloc] init];
[antenna_cfg setLinkProfileIdx:link_profile_idx];
[antenna_cfg setPower:power];
[antenna_cfg setTari:tari];
[antenna_cfg setDoSelect:NO];
error_response = nil;
/* set antenna configuration */
```

```
    result = [_apiInstance srfidSetAntennaConfiguration:_connectedRederID
aAntennaConfiguration:antenna_cfg aStatusMessage:&error_response];

    if (SRFID_RESULT_SUCCESS == result) {
        /* antenna configuration applied successfully */
        NSLog(@"Antenna configuration has been set\n");
    }
    else if (SRFID_RESULT_RESPONSE_ERROR == result) {
        NSLog(@"Error response from RFID reader: %@\n", error_response);
    }
    else if (SRFID_RESULT_RESPONSE_TIMEOUT == result) {
        NSLog(@"Timeout occurs during communication with RFID reader\n");
    }
    else if (SRFID_RESULT_READER_NOT_AVAILABLE == result) {
        NSLog(@"RFID reader with id = %d is not available\n", _connectedRederID);
    }
    else {
        NSLog(@"Request failed\n");
    }
}
```

6.2 Singulation Configuration

Following singulation control settings could be configured via the SDK:

- 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
- Select (SL flag)
- Target (inventory state).

Singulation control settings could be retrieved and set via accordingly *srfidGetSingulationConfiguration* and *srfidSetSingulationConfiguration* API functions as demonstrated in the following example.

```
-(void)getSingulationConfig {
    /* allocate object for storage of singulation settings */
    srfidSingulationConfig *singulation_cfg = [[srfidSingulationConfig alloc] init];

    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;

    /* retrieve singulation configuration */
    SRFID_RESULT result = [_apiInstance srfidGetSingulationConfiguration:_connectedRederID
aSingulationConfig:&singulation_cfg aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        /* singulation configuration received */
        NSLog(@"Tag population: %d\n", [singulation_cfg getTagPopulation]);

        SRFID_SLFLAG slflag = [singulation_cfg getSLFlag];
        switch (slflag) {
            case SRFID_SLFLAG_ASSERTED:
                NSLog(@"SL flag: ASSERTED\n");
                break;
            case SRFID_SLFLAG_DEASSERTED:
                NSLog(@"SL flag: DEASSERTED\n");
                break;
            case SRFID_SLFLAG_ALL:
                NSLog(@"SL flag: ALL\n");
                break;
```

```
    }

    SRFID_SESSION session = [singulation_cfg getSession];
    switch (session) {
      case SRFID_SESSION_S1:
        NSLog(@"Session: S1\n");
        break;
      case SRFID_SESSION_S2:
        NSLog(@"Session: S2\n");
        break;
      case SRFID_SESSION_S3:
        NSLog(@"Session: S3\n");
        break;
      case SRFID_SESSION_S0:
        NSLog(@"Session: S0\n");
        break;
    }

    SRFID_INVENTORYSTATE state = [singulation_cfg getInventoryState];
    switch (state) {
      case SRFID_INVENTORYSTATE_A:
        NSLog(@"Inventory State: State A\n");
        break;
      case SRFID_INVENTORYSTATE_B:
        NSLog(@"Inventory State: State B\n");
        break;
      case SRFID_INVENTORYSTATE_AB_FLIP:
        NSLog(@"Inventory State: AB flip\n");
        break;
    }
  }
}

-(void)setSingulationConfig {
```

```
    /* allocate object for storage of singulation settings */
    srfidSingulationConfig *singulation_cfg = [[srfidSingulationConfig alloc] init];
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
    /* change the received singulation configuration */
        [singulation_cfg setTagPopulation:30];
        [singulation_cfg setSession:SRFID_SESSION_S0];
        [singulation_cfg setSlFlag:SRFID_SLFLAG_ASSERTED];
        [singulation_cfg setInventoryState:SRFID_INVENTORYSTATE_A];
        error_response = nil;
        /* set updated singulation configuration */
    SRFID_RESULT result = [_apiInstance srfidSetSingulationConfiguration:_connectedRederID
aSingulationConfig:singulation_cfg aStatusMessage:&error_response];
        if (SRFID_RESULT_SUCCESS == result) {
            /* singulation configuration applied successfully */
            NSLog(@"Singulation configuration has been set\n");
        }
        else if (SRFID_RESULT_RESPONSE_ERROR == result) {
            NSLog(@"Error response from RFID reader: %@\n", error_response);
        }
        else if (SRFID_RESULT_RESPONSE_TIMEOUT == result) {
            NSLog(@"Timeout occurs during communication with RFID reader\n");
        }
        else if (SRFID_RESULT_READER_NOT_AVAILABLE == result) {
            NSLog(@"RFID reader with id = %d is not available\n", _connectedRederID);
        }
        else {
            NSLog(@"Request failed\n");
        }
}
```

6.3 Trigger Configuration

The SDK provides an ability to configure start and stop trigger parameters. Start trigger parameters include the following:

- Start of an operation based on a physical trigger
- Trigger type (press/release) of a physical trigger
- Delay (in milliseconds) of start of operation
- Repeat monitoring for start trigger after stop of operation.

Start trigger configuration could be retrieved and set via *srfidGetStartTriggerConfiguration* and *srfidSetStartTriggerConfiguration* API functions accordingly.

Stop trigger parameters include the following:

- Stop of an operation based on a physical trigger
- Trigger type (press/release) of a physical trigger
- Stop of an operation based on a specified number of tags inventoried
- Stop of an operation based on a specified timeout (in milliseconds)
- Stop of an operation based on a specified number of inventory rounds completed
- Stop of an operation based on a specified number of access rounds completed.

Stop trigger settings could be retrieved and set via accordingly *srfidGetStopTriggerConfiguration* and *srfidSetStopTriggerConfiguration* API functions.

The following example demonstrates retrieval of current start and stop trigger parameters as well as configuring new start and stop triggers parameters.

```
-(void)getStartTrigger{
    /* allocate object for storage of start trigger settings */
    srfidStartTriggerConfig *start_trigger_cfg = [[srfidStartTriggerConfig alloc] init];
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
    /* retrieve start trigger parameters */
    SRFID_RESULT result = [_apiInstance srfidGetStartTriggerConfiguration:_connectedRederID
aStartTriggeConfig:&start_trigger_cfg aStatusMessage:&error_response];

    if (SRFID_RESULT_SUCCESS == result) {
      /* start trigger configuration received */
      NSLog(@"Start trigger: start on physical trigger = %@\n", ((YES == [start_trigger_cfg
getStartOnHandheldTrigger]) ? @"YES" : @"NO"));
      NSLog(@"Start trigger: physical trigger type = %@\n", ((SRFID_TRIGGERTYPE_PRESS ==
[start_trigger_cfg getTriggerType]) ? @"PRESSED" : @"RELEASED"));
      NSLog(@"Start trigger: delay = %d ms\n", [start_trigger_cfg getStartDelay]);
```

```
        NSLog(@"Start trigger: repeat monitoring = %@\n", ((NO == [start_trigger_cfg getRepeatMonitoring])
? @"NO" : @"YES"));
    }
    else {
        NSLog(@"Failed to receive start trigger parameters\n");
    }
}

-(void)getStoptTrigger {
    //stop
    NSString *error_response = nil;
    /* allocate object for storage of start trigger settings */
    srfidStopTriggerConfig *stop_trigger_cfg = [[srfidStopTriggerConfig alloc] init];
    /* retrieve stop trigger parameters */
    SRFID_RESULT result = [_apiInstance srfidGetStopTriggerConfiguration:_connectedRederID
aStopTriggeConfig:&stop_trigger_cfg aStatusMessage:&error_response];

    if (SRFID_RESULT_SUCCESS == result) {
        /* stop trigger configuration received */
        NSLog(@"Stop trigger: start on physical trigger = %@\n", ((YES == [stop_trigger_cfg
getStopOnHandheldTrigger]) ? @"YES" : @"NO"));
        NSLog(@"Stop trigger: physical trigger type = %@\n", ((SRFID_TRIGGERTYPE_PRESS ==
[stop_trigger_cfg getTriggerType]) ? @"PRESSED" : @"RELEASED"));
        if (YES == [stop_trigger_cfg getStopOnTagCount]) {
            NSLog(@"Stop trigger: stop on %d number of tags received\n", [stop_trigger_cfg
getStopTagCount]);
        }
        if (YES == [stop_trigger_cfg getStopOnTimeout]) {
            NSLog(@"Stop trigger: stop on %d ms timeout\n", [stop_trigger_cfg getStopTimeout]);
        }
        if (YES == [stop_trigger_cfg getStopOnInventoryCount]) {
            NSLog(@"Stop trigger: stop on %d inventory rounds\n", [stop_trigger_cfg getStopInventoryCount]);
        }
```

```
        if (YES == [stop_trigger_cfg getStopOnAccessCount]) {
            NSLog(@"Stop trigger: stop on %d access rounds\n", [stop_trigger_cfg getStopAccessCount]);
        }
    }
    else {
        NSLog(@"Failed to receive stop trigger parameters\n");
    }
}
```

```
-(void)setStopTrigger {
    /* allocate object for storage of start trigger settings */
    srfidStopTriggerConfig *stop_trigger_cfg = [[srfidStopTriggerConfig alloc] init];
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
    /* start on physical trigger */
    [stop_trigger_cfg setStopOnHandheldTrigger:YES];
    [stop_trigger_cfg setTriggerType:SRFID_TRIGGERTYPE_RELEASE];
    [stop_trigger_cfg setStopOnTimeout:YES];
    [stop_trigger_cfg setStopTimout:(5*1000)];
    [stop_trigger_cfg setStopOnTagCount:YES];
    [stop_trigger_cfg setStopTagCount:10];
    [stop_trigger_cfg setStopOnInventoryCount:NO];
    [stop_trigger_cfg setStopOnAccessCount:NO];

    /* set stop trigger parameters */
    SRFID_RESULT result = [_apiInstance srfidSetStopTriggerConfiguration:_connectedRederID
aStopTriggeConfig:stop_trigger_cfg aStatusMessage:&error_response];
```

```
    if (SRFID_RESULT_SUCCESS == result) {
        /* stop trigger configuration applied */
        NSLog(@"Stop trigger configuration has been set\n");
    }
    else {
        NSLog(@"Failed to set stop trigger parameters\n");
    }
}

-(void)setStartTrigger {

    /* allocate object for storage of start trigger settings */
    srfidStartTriggerConfig *start_trigger_cfg = [[srfidStartTriggerConfig alloc] init];
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
    /* configure start trigger parameters */
    /* start on physical trigger */
    [start_trigger_cfg setStartOnHandheldTrigger:YES];
    /* start on physical trigger press */
    [start_trigger_cfg setTriggerType:SRFID_TRIGGERTYPE_PRESS];
    /* repeat monitoring for start trigger conditions after operation stop */
    [start_trigger_cfg setRepeatMonitoring:YES];
    [start_trigger_cfg setStartDelay:0];
    /* set start trigger parameters */
    SRFID_RESULT result = [_apiInstance srfidSetStartTriggerConfiguration:_connectedRederID
aStartTriggeConfig:start_trigger_cfg aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        /* start trigger configuration applied */
        NSLog(@"Start trigger configuration has been set\n");
    }
    else {
        NSLog(@"Failed to set start trigger parameters\n");
    }
```

```
}
```

6.4 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 particular active RFID reader. Supported fields that might be reported include the following:

- First and last seen times
- PC value
- RSSI value
- Phase value
- Channel index
- Tag seen count.

Tag report parameters could be managed via *srfidSetReportConfiguration* and *srfidGetReportConfiguration* API functions as demonstrated in the following example.

```
-(void)getTagReportConfig {
   /* allocate object for storage of tag report settings */
   srfidTagReportConfig *report_cfg = [[srfidTagReportConfig alloc] init];
   /* an object for storage of error response received from RFID reader */
   NSString *error_response = nil;
   /* retrieve tag report parameters */
   SRFID_RESULT result = [_apiInstance srfidGetTagReportConfiguration:_connectedRederID
aTagReportConfig:&report_cfg aStatusMessage:&error_response];
   if (SRFID_RESULT_SUCCESS == result) {
      /* tag report configuration received */
      NSLog(@"PC field: %@\n", ((NO == [report_cfg getIncPC]) ? @"off" : @"on"));
      NSLog(@"Phase field: %@\n", ((NO == [report_cfg getIncPhase]) ? @"off" : @"on"));
      NSLog(@"Channel index field: %@\n", ((NO == [report_cfg getIncChannelIdx]) ? @"off" : @"on"));
      NSLog(@"RSSI field: %@\n", ((NO == [report_cfg getIncRSSI]) ? @"off" : @"on"));
      NSLog(@"Tag seen count field: %@\n", ((NO == [report_cfg getIncTagSeenCount]) ? @"off" : @"on"));
      NSLog(@"First seen time field: %@\n", ((NO == [report_cfg getIncFirstSeenTime]) ? @"off" : @"on"));
      NSLog(@"Last seen time field: %@\n", ((NO == [report_cfg getIncLastSeenTime]) ? @"off" : @"on"));
   }
   else {
      NSLog(@"Failed to receive tag report parameters\n");
   }
```

```
}



-(void)setTagReportConfig {
    /* allocate object for storage of tag report settings */
    srfidTagReportConfig *report_cfg = [[srfidTagReportConfig alloc] init];
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
    /* configure tag report parameters to include only RSSI field */
    [report_cfg setIncRSSI:YES];
    [report_cfg setIncPC:NO];
    [report_cfg setIncPhase:NO];
    [report_cfg setIncChannelIdx:NO];
    [report_cfg setIncTagSeenCount:NO];
    [report_cfg setIncFirstSeenTime:NO];
    [report_cfg setIncLastSeenTime:NO];

    /* set tag report parameters */
    SRFID_RESULT result = [_apiInstance srfidSetTagReportConfiguration:_connectedRederID
aTagReportConfig:report_cfg aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        /* tag report configuration applied */
        NSLog(@"Tag report configuration has been set\n");
    }
    else {
        NSLog(@"Failed to set tag report parameters\n");
    }
}
```

6.5 Regulatory Configuration

The SDK supports managing of regulatory related parameters of a particular active RFID reader. Regulatory configuration includes the following:

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

A set of enabled channels shall include 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 shall not be specified.

Regulatory parameters could be retrieved and set via *srfidGetRegulatoryConfig* and *srfidSetRegulatoryConfig* API functions accordingly. The following example demonstrates retrieving of current regulatory settings and configuring the RFID reader to operate in one of supported regions.

```
-(void)getSetRegulatoryConfig{


    /* allocate object for storage of regulatory settings */

    srfidRegulatoryConfig *regulatory_cfg = [[srfidRegulatoryConfig alloc] init];


    /* an object for storage of error response received from RFID reader */

    NSString *error_response = nil;


    /* retrieve regulatory parameters */

    SRFID_RESULT result = [_apiInstance srfidGetRegulatoryConfig:_connectedRederID
aRegulatoryConfig:&regulatory_cfg aStatusMessage:&error_response];

    if (SRFID_RESULT_SUCCESS == result) {
```

```
        /* regulatory configuration received */
        if (NSOrderedSame == [[regulatory_cfg getRegionCode] caseInsensitiveCompare:@"NA"]) {
            NSLog(@"Regulatory: region is NOT set\n");
        }
        else {
            NSLog(@"Region code: %@\n", [regulatory_cfg getRegionCode]);
            SRFID_HOPPINGCONFIG hopping_cfg = [regulatory_cfg getHoppingConfig];
            NSLog(@"Hopping is %@\n", ((SRFID_HOPPINGCONFIG_DISABLED == hopping_cfg) ? @"off" :
@"on"));
            NSArray *channels = [regulatory_cfg getEnabledChannelsList];
            for (NSString *str in channels) {
                NSLog(@"Enabled channel: %@\n", str);
            }
        }
    }
    else {
        NSLog(@"Failed to receive regulatory parameters\n");
    }
    /* code of region to be set as current one */
    NSString *region_code = @"USA";
    /* an array of enabled channels to be set */
    NSMutableArray *enabled_channels = [[NSMutableArray alloc] init];
    /* a hopping to be set  */
    SRFID_HOPPINGCONFIG hopping_on = SRFID_HOPPINGCONFIG_DISABLED;

    /* allocate object for storage of region information */
    NSMutableArray *regions = [[NSMutableArray alloc] init];

    /* retrieve supported regions */
    result = [_apiInstance srfidGetSupportedRegions:_connectedRederID aSupportedRegions:&regions
aStatusMessage:&error_response];

    if (SRFID_RESULT_SUCCESS == result) {
```

```
        /* supported regions information received */
        /* select the last supported regions to be set as current one */
        region_code = [NSString stringWithFormat:@"%@", [(srfidRegionInfo*)[regions lastObject]
getRegionCode]];
    }

    /* allocate object for storage of supported channels information */
    NSMutableArray *supported_channels = [[NSMutableArray alloc] init];
    BOOL hopping_configurable = NO;

    /* retrieve detailed information about region specified by region code */
    result = [_apiInstance srfidGetRegionInfo:_connectedRederID aRegionCode:region_code
aSupportedChannels:&supported_channels aHoppingConfigurable:&hopping_configurable
aStatusMessage:&error_response];

    if (SRFID_RESULT_SUCCESS == result) {
        /* region information received */

        if (YES == hopping_configurable) {
            /* region supports hopping */
            /* enable first and last channels from the set of supported channels */
            [enabled_channels addObject:[supported_channels firstObject]];
            [enabled_channels addObject:[supported_channels lastObject]];
            /* enable hopping */
            hopping_on = SRFID_HOPPINGCONFIG_ENABLED;
        }
        else {
            /* region does not support hopping */
            /* request to not configure hopping */
            hopping_on = SRFID_HOPPINGCONFIG_DEFAULT;
        }
    }
    error_response = nil;
```

```
    /* configure regulatory parameters to be set */
    regulatory_cfg = [[srfidRegulatoryConfig alloc] init];
    [regulatory_cfg setRegionCode:region_code];
    [regulatory_cfg setEnabledChannelsList:enabled_channels];
    [regulatory_cfg setHoppingConfig:hopping_on];

    /* set regulatory parameters */
    result = [_apiInstance srfidSetRegulatoryConfig:_connectedRederID aRegulatoryConfig:regulatory_cfg
aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        /* regulatory configuration applied */
        NSLog(@"Tag report configuration has been set\n");
    }
    else if (SRFID_RESULT_RESPONSE_ERROR == result) {
        NSLog(@"Error response from RFID reader: %@\n", error_response);
    }
    else {
        NSLog(@"Failed to set regulatory parameters\n");
    }

}
```

6.6 Pre-filters Configuration

Pre-filters are same as the select command of C1G2 specification. The SDK supports pre-filters configuration of a particular active RFID reader. When pre-filters are configured, they could be applied prior to inventory operations.

Following parameters could be configured for each pre-filter:

- Target (Session S0, Session S1, Session S2, Session S3, Select Flag)
- Action
- Memory bank (epc, tid, user)
- Mask start position (in words): indicates start position from beginning of memory bank from were match pattern is checked
- Match pattern.

Configured pre-filters could be retrieved from a particular active RFID reader via *srfidGetPreFilters* API function. The *srfidSetPreFilters* API function is used to configure a new set of pre-filters. The following example demonstrates pre-filters management supported by the SDK.

```
-(void)getSetPrefilterConfig{
    /* allocate object for storage of pre filters */
    NSMutableArray *prefilters = [[NSMutableArray alloc] init];
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
    /* retrieve pre-filters */
    SRFID_RESULT result = [_apiInstance srfidGetPreFilters:_connectedRederID aPreFilters:&prefilters
aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        /* pre-filters received */
        NSLog(@"Number of pre-filters: %lu\n", (unsigned long)[prefilters count]);

        for (srfidPreFilter *filter in prefilters) {
            NSLog(@"Match pattern: %@\n", [filter getMatchPattern]);
            NSLog(@"Mask start position: %d words\n", [filter getMaskStartPos]);

            SRFID_SELECTACTION action = [filter getAction];
            switch (action) {
                case SRFID_SELECTACTION_INV_A2BB2A_NOT_INV_A__OR__NEG_SL_NOT_ASRT_SL:
                    NSLog(@"Action: INV A2BB2A NOT INV A OR NEG SL NOT ASRT SL\n");
                    break;
                case SRFID_SELECTACTION_INV_A__OR__ASRT_SL:
                    NSLog(@"Action: INV A OR ASRT SL\n");
                    break;
                case SRFID_SELECTACTION_INV_A_NOT_INV_B__OR__ASRT_SL_NOT_DSRT_SL:
                    NSLog(@"Action: INV A NOT INV B OR ASRT SL NOT DSRT SL\n");
                    break;
                case SRFID_SELECTACTION_INV_B__OR__DSRT_SL:
                    NSLog(@"Action: INV B OR DSRT SL\n");
                    break;
                case SRFID_SELECTACTION_INV_B_NOT_INV_A__OR__DSRT_SL_NOT_ASRT_SL:
                    NSLog(@"Action: INV B NOT INV A OR DSRT SL NOT ASRT SL\n");
```

```
        break;
    case SRFID_SELECTACTION_NOT_INV_A2BB2A__OR__NOT_NEG_SL:
        NSLog(@"Action: NOT INV A2BB2A OR NOT NEG SL\n");
        break;
    case SRFID_SELECTACTION_NOT_INV_A__OR__NOT_ASRT_SL:
        NSLog(@"Action: NOT INV A OR NOT ASRT SL\n");
        break;
    case SRFID_SELECTACTION_NOT_INV_B__OR__NOT_DSRT_SL:
        NSLog(@"Action: NOT INV B OR NOT DSRT SL\n");
        break;
}

SRFID_SELECTTARGET target = [filter getTarget];
switch (target) {
    case SRFID_SELECTTARGET_S0:
        NSLog(@"Target: Session SO\n");
        break;
    case SRFID_SELECTTARGET_S1:
        NSLog(@"Target: Session S1\n");
        break;
    case SRFID_SELECTTARGET_S2:
        NSLog(@"Target: Session S2\n");
        break;
    case SRFID_SELECTTARGET_S3:
        NSLog(@"Target: Session S3\n");
        break;
    case SRFID_SELECTTARGET_SL:
        NSLog(@"Target: Select Flag\n");
        break;
}

SRFID_MEMORYBANK bank = [filter getMemoryBank];
switch (bank) {
```

```
            case SRFID_MEMORYBANK_EPC:
                NSLog(@"Memory Bank: EPC\n");
                break;
            case SRFID_MEMORYBANK_RESV:
                NSLog(@"Memory Bank: RESV\n");
                break;
            case SRFID_MEMORYBANK_TID:
                NSLog(@"Memory Bank: TID\n");
                break;
            case SRFID_MEMORYBANK_USER:
                NSLog(@"Memory Bank: USER\n");
                break;
            case SRFID_MEMORYBANK_NONE:
                NSLog(@"MEMORY BANK NONE\n");
                break;
            case SRFID_MEMORYBANK_ACCESS:
                NSLog(@"MEMORY BANK ACCESS\n");
                break;
            case SRFID_MEMORYBANK_KILL:
                NSLog(@"MEMORY BANK KILL\n");
                break;
            case SRFID_MEMORYBANK_ALL:
                NSLog(@"MEMORY BANK ALL\n");
                break;
        }
    }
}
else {
    NSLog(@"Failed to receive pre-filters\n");
}
[prefilters removeAllObjects];

/* create one pre-filter */
```

```
    srfidPreFilter *filter = [[srfidPreFilter alloc] init];
    [filter setMatchPattern:@"N20122014R1010364989126V"];
    [filter setMaskStartPos:2];
    [filter setMemoryBank:SRFID_MEMORYBANK_EPC];
    [filter setAction:SRFID_SELECTACTION_INV_A__OR__ASRT_SL];
    [filter setTarget:SRFID_SELECTTARGET_SL];

    [prefilters addObject:filter];
    error_response = nil;

    /* set pre-filters */
    result = [_apiInstance srfidSetPreFilters:_connectedRederID aPreFilters:prefilters
aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        /* pre-filters have been set */
        NSLog(@"Pre-filters has been set\n");
    }
    else if (SRFID_RESULT_RESPONSE_ERROR == result) {
        NSLog(@"Error response from RFID reader: %@\n", error_response);
    }
    else {
        NSLog(@"Failed to set tag report parameters\n");
    }
}
```

6.7 Beeper Configuration

The SDK provides an ability to configure a beeper of a particular active RFID reader. The beeper could be configured to one of predefined volumes (low, medium, high) or be disabled. Retrieving and setting

of beeper configuration is performed via *srfidSetBeeperConfig* and *srfidGetBeeperConfig* API functions as demonstrated in the following example.

```
-(void)getSetBeeperConfig{
    /* object for beeper configuration */
    SRFID_BEEPERCONFIG beeper_cfg;

    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;

    /* retrieve beeper configuration */
    SRFID_RESULT result = [_apiInstance srfidGetBeeperConfig:_connectedRederID
aBeeperConfig:&beeper_cfg aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        /* beeper configuration received */
        switch (beeper_cfg) {
            case SRFID_BEEPERCONFIG_HIGH:
                NSLog(@"Beeper: high volume\n");
                break;
            case SRFID_BEEPERCONFIG_LOW:
                NSLog(@"Beeper: low volume\n");
                break;
            case SRFID_BEEPERCONFIG_MEDIUM:
                NSLog(@"Beeper: medium volume\n");
                break;
            case SRFID_BEEPERCONFIG_QUIET:
                NSLog(@"Beeper: disabled\n");
                break;
        }
    }
    else {
        NSLog(@"Failed to receive beeper parameters\n");
    }
    error_response = nil;
```

```
    /* disable beeper */
    result = [_apiInstance srfidSetBeeperConfig:_connectedRederID
aBeeperConfig:SRFID_BEEPERCONFIG_QUIET aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        /* beeper configuration applied */
        NSLog(@"Beeper configuration has been set\n");
    }
    else if (SRFID_RESULT_RESPONSE_ERROR == result) {
        NSLog(@"Error response from RFID reader: %@\n", error_response);
    }
    else {
        NSLog(@"Failed to set beeper configuration\n");
    }
}
```

6.8 Managing Configuration

Various parameter of a particular RFID reader configured via SDK are lost after next power down. The SDK provides an ability to store and restore a persistent configuration of RFID reader. The *srfidSaveReaderConfiguration* API function could be used to either make current configuration persistent over power down and power up cycles or store current configuration to custom defaults area. The configuration stored to custom defaults area could be restored via *srfidRestoreReaderConfiguration* API function. The same API function is used to restore the factory defined configuration. The following example demonstrates utilization of mentioned API functions.

```
-(void)saveReaderCurrentConfigurationPersistent {
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;

    /* cause the RFID reader to make current configuration persistent */
    SRFID_RESULT result = [_apiInstance srfidSaveReaderConfiguration:_connectedRederID
aSaveCustomDefaults:NO aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        NSLog(@"Current configuration became persistent\n");
```

```
    }
    else {
        NSLog(@"Request failed\n");
    }
}



-(void)saveReaderCurrentConfigurationWithCustomDefaults {
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
    /* cause the RFID reader to save current configuration in custom defaults area */
    SRFID_RESULT result = [_apiInstance srfidSaveReaderConfiguration:_connectedRederID
aSaveCustomDefaults:YES aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        NSLog(@"Current configuration stored in custom defaults\n");
    }
    else {
        NSLog(@"Request failed\n");
    }
}



-(void)restoreReaderConfigurationFromCustomDefaults {

    /* an object for storage of error response received from RFID reader */
```

```
    NSString *error_response = nil;
    /* cause the RFID reader to restore configuration from custom defaults */
    SRFID_RESULT result = [_apiInstance srfidRestoreReaderConfiguration:_connectedRederID
aRestoreFactoryDefaults:NO aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        NSLog(@"Request Success\n");
    }
}



-(void)restoreReaderConfigurationWithFactoryDefinedConfiguration {

    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
    /* cause the RFID reader to restore factory defined configuration*/
    SRFID_RESULT result = [_apiInstance srfidRestoreReaderConfiguration:_connectedRederID
aRestoreFactoryDefaults:YES aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        NSLog(@"Request Success\n");
    }

}
```

7. Performing Operations

Zebra Bluetooth RFID iOS SDK API enables performing various radio operations with a particular active RFID reader.

7.1 Rapid Read

Rapid read operation is a simple inventory operation without performing a read from a particular memory bank.

The *srfidStartRapidRead* API function is used to request performing of rapid read operation. Aborting of rapid read operation is requested via *srfidStopRapidRead* API function. When performing of rapid read operation is requested the actual operation will be started once conditions specified by start trigger parameters are met. The on-going operation will be stopped in accordance with configured stop trigger parameters. If repeat monitoring option is enabled in start trigger configuration the actual operation will be started again after it has stopped once conditions of start trigger configuration are met. On starting and stopping of the actual operation the SDK will deliver asynchronous notifications to the application if the application has subscribed for events of this type.

The SDK will deliver asynchronous notifications to inform the application about tag data received from the RFID reader during the on-going operation if the application has subscribed for events of this type. Fields to be reported during asynchronous tag data related notification are configured via *reportConfig* parameter of *srfidStartRapidRead* API function.

The following example demonstrates performing of rapid read operation that starts and stops immediately after requested operation performing and aborting.

```
-(void)startStopRapidRead{

    /* subscribe for tag data related events */
    [apiInstance srfidSubsribeForEvents:SRFID_EVENT_MASK_READ];
    /* subscribe for operation start/stop related events */
    [apiInstance srfidSubsribeForEvents:SRFID_EVENT_MASK_STATUS];

    /* allocate object for start trigger settings */
    srfidStartTriggerConfig *start_trigger_cfg = [[srfidStartTriggerConfig alloc] init];
    /* allocate object for stop trigger settings */
    srfidStopTriggerConfig *stop_trigger_cfg = [[srfidStopTriggerConfig alloc] init];

    /* allocate object for report parameters of rapid read operation */
    srfidReportConfig *report_cfg = [[srfidReportConfig alloc] init];
```

```
    /* allocate object for access parameters of rapid read operation */
    srfidAccessConfig *access_cfg = [[srfidAccessConfig alloc] init];

    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
    do {
        /* configure start and stop triggers parameters to start and stop actual operation immediately on a
corresponding response */
        [start_trigger_cfg setStartOnHandheldTrigger:NO];
            [start_trigger_cfg setStartDelay:0];
            [start_trigger_cfg setRepeatMonitoring:NO];

            [stop_trigger_cfg setStopOnHandheldTrigger:NO];
            [stop_trigger_cfg setStopOnTimeout:NO];
            [stop_trigger_cfg setStopOnTagCount:NO];
            [stop_trigger_cfg setStopOnInventoryCount:NO];
            [stop_trigger_cfg setStopOnAccessCount:NO];
        /* set start trigger parameters */
            SRFID_RESULT result = [apiInstance srfidSetStartTriggerConfiguration:connectedReaderId
aStartTriggeConfig:start_trigger_cfg aStatusMessage:&error_response];
            if (SRFID_RESULT_SUCCESS == result) {
                /* start trigger configuration applied */
                NSLog(@"Start trigger configuration has been set\n");
            }
            else {
                NSLog(@"Failed to set start trigger parameters\n");
                break;
            }
        /* set stop trigger parameters */
            result = [apiInstance srfidSetStopTriggerConfiguration:connectedReaderId
aStopTriggeConfig:stop_trigger_cfg aStatusMessage:&error_response];
            if (SRFID_RESULT_SUCCESS == result) {
```

```
            /* stop trigger configuration applied */
            NSLog(@"Stop trigger configuration has been set\n");
        }
        else {
            NSLog(@"Failed to set stop trigger parameters\n");
            break;
        }

    /* start and stop triggers have been configured */
        error_response = nil;

        /* configure report parameters to report RSSI, Channel Index, Phase and PC fields */
        [report_cfg setIncPC:YES];
        [report_cfg setIncPhase:YES];
        [report_cfg setIncChannelIndex:YES];
        [report_cfg setIncRSSI:YES];
        [report_cfg setIncTagSeenCount:NO];
        [report_cfg setIncFirstSeenTime:NO];
        [report_cfg setIncLastSeenTime:NO];

        /* configure access parameters to perform the operation with 27.0 dbm antenna power level without
application of pre-filters */
        [access_cfg setPower:270];
        [access_cfg setDoSelect:NO];


    /* request performing of rapid read operation */
    result = [apiInstance srfidStartRapidRead:connectedReaderId aReportConfig:report_cfg
aAccessConfig:access_cfg aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        NSLog(@"Request succeed\n");

        /* stop an operation after 1 minute */
```

```
            dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(60 * NSEC_PER_SEC)),
dispatch_get_main_queue(), ^{
                [self->apiInstance srfidStopRapidRead:self->connectedReaderId aStatusMessage:nil];
            });
        }
        else if (SRFID_RESULT_RESPONSE_ERROR == result) {
            NSLog(@"Error response from RFID reader: %@\n", error_response);
        }
        else {
            NSLog(@"Request failed\n");
        }

    } while (0);

}
```

Events

```
- (void)srfidEventStatusNotify:(int)readerID aEvent:(SRFID_EVENT_STATUS)event
aNotification:(id)notificationData {
    NSLog(@"Radio operation has %@\n", ((SRFID_EVENT_STATUS_OPERATION_START == event) ?
@"started" : @"stopped"));
}

- (void)srfidEventReadNotify:(int)readerID aTagData:(srfidTagData *)tagData {
    /* print the received tag data */
    NSLog(@"Tag data received from RFID reader with ID = %d\n", readerID);
    NSLog(@"Tag id: %@\n", [tagData getTagId]);

}
```

7.2 Inventory

Inventory is an advanced inventory operation being performed simultaneously with reading from a particular memory bank.

Inventory operation is performed similarly to the rapid read operation described above. Thus performing and aborting of the inventory operation is requested through *srfidStartInventory* and *srfidStopInventory* API functions accordingly. After request of operation performing the actual operation will be started in accordance with the configured start trigger parameters and will be stopped once conditions specified by stop trigger parameters are met. After the operation has stopped it might be started again if it is not aborted and the repeat monitoring option is enabled in start trigger configuration. The SDK informs the application about starting and stopping of the actual notification through corresponding asynchronous notifications.

The SDK will deliver asynchronous notifications to inform the application about tag data received from the RFID reader during the on-going operation if the application has subscribed for events of this type. Fields to be reported during asynchronous tag data related notification are configured via *reportConfig* parameter of *srfidStartInventory* API function.

The following example demonstrates performing of a continuous inventory operation with reading from EPC memory bank that starts on a press of a physical trigger and stops on a release of a physical trigger or after a 25 second timeout.

```
-(void)startStopInventory{

    /* subscribe for tag data related events */
    [apiInstance srfidSubsribeForEvents:SRFID_EVENT_MASK_READ];
    /* subscribe for operation start/stop related events */
    [apiInstance srfidSubsribeForEvents:SRFID_EVENT_MASK_STATUS];

    /* identifier of one of active RFID readers is supposed to be stored in m_ReaderId variable */

    /* allocate object for start trigger settings */
```

```
    srfidStartTriggerConfig *start_trigger_cfg = [[srfidStartTriggerConfig alloc] init];

    /* allocate object for stop trigger settings */
    srfidStopTriggerConfig *stop_trigger_cfg = [[srfidStopTriggerConfig alloc] init];

    /* allocate object for report parameters of inventory operation */
    srfidReportConfig *report_cfg = [[srfidReportConfig alloc] init];

    /* allocate object for access parameters of inventory operation */
    srfidAccessConfig *access_cfg = [[srfidAccessConfig alloc] init];

    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;

    do {
        /* configure start triggers parameters to start on physical trigger press */
        [start_trigger_cfg setStartOnHandheldTrigger:YES];
        [start_trigger_cfg setTriggerType:SRFID_TRIGGERTYPE_PRESS];
        [start_trigger_cfg setStartDelay:0];
        [start_trigger_cfg setRepeatMonitoring:YES];

        /* configure stop triggers parameters to stop on physical trigger release or on 25 sec timeout  */
        [stop_trigger_cfg setStopOnHandheldTrigger:YES];
        [stop_trigger_cfg setTriggerType:SRFID_TRIGGERTYPE_RELEASE];
        [stop_trigger_cfg setStopOnTimeout:YES];
        [stop_trigger_cfg setStopTimout:(25*1000)];
        [stop_trigger_cfg setStopOnTagCount:NO];
        [stop_trigger_cfg setStopOnInventoryCount:NO];
        [stop_trigger_cfg setStopOnAccessCount:NO];

        /* set start trigger parameters */
        SRFID_RESULT result = [apiInstance srfidSetStartTriggerConfiguration:connectedReaderId
aStartTriggeConfig:start_trigger_cfg aStatusMessage:&error_response];
```

```
    if (SRFID_RESULT_SUCCESS == result) {
        /* start trigger configuration applied */
        NSLog(@"Start trigger configuration has been set\n");
    }
    else {
        NSLog(@"Failed to set start trigger parameters\n");
        break;
    }
    /* set stop trigger parameters */
    result = [apiInstance srfidSetStopTriggerConfiguration:connectedReaderId
aStopTriggeConfig:stop_trigger_cfg aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        /* stop trigger configuration applied */
        NSLog(@"Stop trigger configuration has been set\n");
    }
    else {
        NSLog(@"Failed to set stop trigger parameters\n");
        break;
    }

    /* start and stop triggers have been configured */
    error_response = nil;

    /* configure report parameters to report RSSI and Channel Index fields */
    [report_cfg setIncPC:NO];
    [report_cfg setIncPhase:NO];
    [report_cfg setIncChannelIndex:YES];
    [report_cfg setIncRSSI:YES];
    [report_cfg setIncTagSeenCount:NO];
    [report_cfg setIncFirstSeenTime:NO];
    [report_cfg setIncLastSeenTime:NO];
```

```
        /* configure access parameters to perform the operation with 27.0 dbm antenna power level without
application of pre-filters */
        [access_cfg setPower:270];
        [access_cfg setDoSelect:NO];
        /* request performing of inventory operation with reading from EPC memory bank */
        result = [apiInstance srfidStartInventory:connectedReaderId
aMemoryBank:SRFID_MEMORYBANK_EPC aReportConfig:report_cfg aAccessConfig:access_cfg
aStatusMessage:&error_response];

        if (SRFID_RESULT_SUCCESS == result) {
            NSLog(@"Request succeed\n");
            /* request abort of an operation after 1 minute */
            dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(60 * NSEC_PER_SEC)),
dispatch_get_main_queue(), ^{
                [self->apiInstance srfidStopInventory:self->connectedReaderId aStatusMessage:nil];
            });
        }
        else if (SRFID_RESULT_RESPONSE_ERROR == result) {
            NSLog(@"Error response from RFID reader: %@\n", error_response);
        }
        else {
            NSLog(@"Request failed\n");
        }

    } while (0);


}
```

Events

```
- (void)srfidEventStatusNotify:(int)readerID aEvent:(SRFID_EVENT_STATUS)event
aNotification:(id)notificationData {
    NSLog(@"Radio operation has %@\n", ((SRFID_EVENT_STATUS_OPERATION_START == event) ?
@"started" : @"stopped"));
```

```
}
- (void)srfidEventReadNotify:(int)readerID aTagData:(srfidTagData *)tagData {
    /* print the received tag data */
    NSLog(@"Tag data received from RFID reader with ID = %d\n", readerID);
    NSLog(@"Tag id: %@\n", [tagData getTagId]);
    SRFID_MEMORYBANK bank = [tagData getMemoryBank];
    if (SRFID_MEMORYBANK_NONE != bank) {
        NSString *str_bank = @"";
        switch (bank) {
            case SRFID_MEMORYBANK_EPC:
                str_bank = @"EPC";
                break;
            case SRFID_MEMORYBANK_TID:
                str_bank = @"TID";
                break;
            case SRFID_MEMORYBANK_USER:
                str_bank = @"USER";
                break;

            case SRFID_MEMORYBANK_RESV:
                str_bank = @"RESV";
                break;
            case SRFID_MEMORYBANK_NONE:
                str_bank = @"None";
                break;
            case SRFID_MEMORYBANK_ACCESS:
                str_bank = @"Acess";
                break;
            case SRFID_MEMORYBANK_KILL:
                str_bank = @"Kill";
                break;
            case SRFID_MEMORYBANK_ALL:
                str_bank = @"All";
```

```
            break;
        }
        NSLog(@"%@ memory bank data: %@\n", str_bank, [tagData getMemoryBankData]);
    }
}
```

7.3 Inventory with Pre-filters

If pre-filters are configured they might be applied during performing of inventory operation. Application of pre-filters is enabled via *accessConfig* parameter of *srfidStartInventory* and *srfidStartRapidRead* API functions. Excepting enablement of pre-filters application in *accessConfig* parameter inventory with pre-filters is performed similarly to a typical inventory operation described above. The following example demonstrates enabling application of configured pre-filters during inventory operation.

```
-(void)startStopInventoryWithPrefilters {
    /* allocate object for report parameters of inventory operation */
    srfidReportConfig *report_cfg = [[srfidReportConfig alloc] init];

    /* allocate object for access parameters of inventory operation */
    srfidAccessConfig *access_cfg = [[srfidAccessConfig alloc] init];

    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;

    /* configure report parameters to report RSSI field */
    [report_cfg setIncPC:NO];
    [report_cfg setIncPhase:NO];
    [report_cfg setIncChannelIndex:NO];
    [report_cfg setIncRSSI:YES];
    [report_cfg setIncTagSeenCount:NO];
    [report_cfg setIncFirstSeenTime:NO];
    [report_cfg setIncLastSeenTime:NO];

    /* configure access parameters to perform the operation with 27.0 dbm antenna power level */
```

```
    [access_cfg setPower:270];
    /* enable application of configured pre-filters */
    [access_cfg setDoSelect:YES];

    /* request performing of inventory operation with reading from EPC memory bank */
    SRFID_RESULT result = [apiInstance srfidStartInventory:connectedReaderId
aMemoryBank:SRFID_MEMORYBANK_EPC aReportConfig:report_cfg aAccessConfig:access_cfg
aStatusMessage:&error_response];

    if (SRFID_RESULT_SUCCESS == result) {
        NSLog(@"Request succeed\n");

        /* request abort of an operation after 1 minute */

        dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(60 * NSEC_PER_SEC)),
dispatch_get_main_queue(), ^{
            [self->apiInstance srfidStopInventory:self->connectedReaderId aStatusMessage:nil];
        });
    }
    else if (SRFID_RESULT_RESPONSE_ERROR == result) {
        NSLog(@"Error response from RFID reader: %@\n", error_response);
    }
    else {
        NSLog(@"Request failed\n");
    }

}
```

7.4 Tag Locationing

The SDK provides an ability to perform tag locationing operation. The *srfidStartTagLocationing* API function is used to request performing of tag locationing operation. Aborting of tag locationing operation is requested via *srfidStopTagLocationing* API function. The actual operation is started and stopped based on configured start and stop triggers parameters. The SDK informs the application about starting and stopping of the actual operation via delivery of asynchronous notifications if the

application has subscriber for events of this type. During an on-going operation the SDK will deliver asynchronous notifications to inform the application about current tag proximity value (in percents).

The following example demonstrates performing of tag locationing operation.

```
-(void)tagLocationing{


    /* subscribe for tag locationing related events */
    [apiInstance srfidSubsribeForEvents:SRFID_EVENT_MASK_PROXIMITY];
    /* subscribe for operation start/stop related events */
    [apiInstance srfidSubsribeForEvents:SRFID_EVENT_MASK_STATUS];
    /* identifier of one of active RFID readers is supposed to be stored in m_ReaderId variable */
    /* id of tag to be located */
    NSString *tag_id = @"V6219894630101R41022102N";
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;
    SRFID_RESULT result = [apiInstance srfidStartTagLocationing:connectedReaderId aTagEpcId:tag_id
aStatusMessage:&error_response];
    if (SRFID_RESULT_SUCCESS == result) {
        NSLog(@"Request succeed\n");
        /* request abort of an operation after 1 minute */
        dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(60 * NSEC_PER_SEC)),
dispatch_get_main_queue(), ^{
            [self->apiInstance srfidStopTagLocationing:self->connectedReaderId aStatusMessage:nil];
        });
    }
    else if (SRFID_RESULT_RESPONSE_ERROR == result) {
        NSLog(@"Error response from RFID reader: %@\n", error_response);
    }
    else {
        NSLog(@"Request failed\n");
    }

}
```

7.5 Multi Tag Locationing

By using this API, users can do multi-tag locationing.

```
/// Start multi tag locationing
-(void)startMultiTagLocationing {

    NSString *error_response = nil;

    srfidReportConfig *multipleTagsReportConfig;

    NSString *tag_id_1 = @"36420124102N012610R98V91";
    NSString *tag_id_2 = @"211241451351513251351324";
    NSString *tag_id_3 = @"434563463462345623456346";

    [multipleTagsReportConfig addItem:tag_id_1 aRSSIValueLimit:-(40)];
    [multipleTagsReportConfig addItem:tag_id_2 aRSSIValueLimit:-(40)];
    [multipleTagsReportConfig addItem:tag_id_3 aRSSIValueLimit:-(40)];

    SRFID_RESULT result = [_apiInstance srfidStartMultiTagsLocationing:_connectedRederID
aReportConfig:multipleTagsReportConfig aAccessConfig:nil aStatusMessage:&error_response];

    if (SRFID_RESULT_SUCCESS == result) {
        NSLog(@"Request succeed\n");
    }else{
        NSLog(@"Request failed\n");
    }
}
```

```
/// Stop multi tag locationing
-(void)stopMultiTagLocationing{

    NSString *error_response = nil;

    SRFID_RESULT result = [_apiInstance srfidStopMultiTagsLocationing:_connectedRederID
aStatusMessage:&error_response];

    if (SRFID_RESULT_SUCCESS == result) {
        NSLog(@"Request succeed\n");
    }else{
        NSLog(@"Request failed\n");
    }
}
```

7.6 Access Operations

The SDK supports performing of read, write, lock and kill access operations on a specific tag. Access operations are performed via *srfidReadTag*, *srfidWriteTag*, *srfidLockTag* and *srfiKillTag* API functions accordingly. The mentioned API functions are performed synchronously; the corresponding operation is started immediately and is stopped once tag data is reported by RFID reader or after a 5 seconds timeout.

The following example demonstrates performing of read and write access operations on one of tags being inventoried.

```
-(void)accessOperationTagReadAndWrite {


    /* allocate object for storing results of access operation */
    srfidTagData *access_result = [[srfidTagData alloc] init];
    /* id of tag to be read */
    NSString *tag_id = @"36420124102N012610R98V91";
    /* an object for storage of error response received from RFID reader */
    NSString *error_response = nil;


    /* request to read 8 words from EPC memory bank of tag specified by tag _id */
    SRFID_RESULT result = [apiInstance srfidReadTag:connectedReaderId aTagID:tag_id
aAccessTagData:&access_result aMemoryBank:SRFID_MEMORYBANK_EPC aOffset:0 aLength:8
aPassword:0x00 aStatusMessage:&error_response];
```

```
    if (SRFID_RESULT_SUCCESS == result) {
        NSLog(@"Request succeed\n");

        /* check result code of access operation */
        if (NO == [access_result getOperationSucceed]) {
            NSLog(@"Read operation has failed with error: %@\n", [access_result getOperationStatus]);
        }
        else {
            NSLog(@"Memory bank data: %@", [access_result getMemoryBankData]);
        }
    }
    else if (SRFID_RESULT_RESPONSE_ERROR == result) {
        NSLog(@"Error response from RFID reader: %@\n", error_response);
    }
    else if (SRFID_RESULT_RESPONSE_TIMEOUT == result) {
        NSLog(@"Timeout occurred\n");
    }
    else {
        NSLog(@"Request failed\n");
    }

    access_result = [[srfidTagData alloc] init];
    error_response = nil;

    /* data to be written */
    NSString *data = @"N20122014R1010364989126V";

    /* request to write a data to a EPC memory bank of tag specified by tag_id */
    result = [apiInstance srfidWriteTag:connectedReaderId aTagID:tag_id aAccessTagData:&access_result
aMemoryBank:SRFID_MEMORYBANK_EPC aOffset:0 aData:data aPassword:0x00 aDoBlockWrite:NO
aStatusMessage:&error_response];

    if (SRFID_RESULT_SUCCESS == result) {
```

```
        NSLog(@"Request succeed\n");

        /* check result code of access operation */
        if (NO == [access_result getOperationSucceed]) {
            NSLog(@"Write operation has failed with error: %@\n", [access_result getOperationStatus]);
        }
    }
    else if (SRFID_RESULT_RESPONSE_ERROR == result) {
        NSLog(@"Error response from RFID reader: %@\n", error_response);
    }
    else if (SRFID_RESULT_RESPONSE_TIMEOUT == result) {
        NSLog(@"Timeout occurred\n");
    }
    else {
        NSLog(@"Request failed\n");
    }

}
```

7.7 Gen2V2 Untraceable API

7.7.1 srfidAuthenticate:

Access criteria allow us to set up the filters for the inventory operation.

```
    // initialize access criteria
    srfidAccessCriteria *accessCriteria = [[srfidAccessCriteria alloc] init];
    // setup tag filter 1
    srfidTagFilter *tagFilter1 = [[srfidTagFilter alloc] init];
    [tagFilter1 setFilterMaskBank:SRFID_MEMORYBANK_EPC];
    [tagFilter1 setFilterData:@"0176"];
    [tagFilter1 setFilterDoMatch:YES];
    [tagFilter1 setFilterMask:@"FFFF"];
```

```
    [tagFilter1 setFilterMaskStartPos:2];
    [tagFilter1 setFilterMatchLength:1];
    // set tag filter 1
    [accessCriteria setTagFilter1:tagFilter1];
```

7.7.2 srfidUntraceable:

```
srfidUntraceableConfig *untraceConfig = [[srfidUntraceableConfig alloc]init];
    NSString *status = [[NSString alloc]init];
    [untraceConfig setShowEpc:NO];
    [untraceConfig setEpcLen:2];
    [untraceConfig setShowUser:YES];
    [untraceConfig setTid:SRFID_TID_SHOW];
    SRFID_RESULT result = [_apiInstance srfidUntraceable:self->_connectedRederID
aAccessCriteria:accessCriteria aAccessConfig:nil
    aPassword:01 aUntraceableConfig:untraceConfig aStatusMessage:&status];
```

Sample code for Untraceable API:

```
#pragma mark - Methods - Set Untraceable attribute

-(void)setUntraceable {

    // initialize access criteria
    srfidAccessCriteria *accessCriteria = [[srfidAccessCriteria alloc] init];
    // setup tag filter 1
    srfidTagFilter *tagFilter1 = [[srfidTagFilter alloc] init];
    [tagFilter1 setFilterMaskBank:SRFID_MEMORYBANK_EPC];
```

```
    [tagFilter1 setFilterData:@"0176"];
    [tagFilter1 setFilterDoMatch:YES];
    [tagFilter1 setFilterMask:@"FFFF"];
    [tagFilter1 setFilterMaskStartPos:2];
    [tagFilter1 setFilterMatchLength:1];
    // set tag filter 1
    [accessCriteria setTagFilter1:tagFilter1];



    srfidUntraceableConfig *untraceConfig = [[srfidUntraceableConfig alloc]init];
    NSString *status = [[NSString alloc]init];
    [untraceConfig setShowEpc:NO];
    [untraceConfig setEpcLen:2];
    [untraceConfig setShowUser:YES];
    [untraceConfig setTid:SRFID_TID_SHOW];
    SRFID_RESULT result = [_apiInstance srfidUntraceable:self->_connectedRederID
aAccessCriteria:accessCriteria aAccessConfig:nil
    aPassword:01 aUntraceableConfig:untraceConfig aStatusMessage:&status];

    if (SRFID_RESULT_SUCCESS == result) {
        NSLog(@"Set Untraceable Success");
    }else {
        NSLog(@"Failed to set untraceable");
    }
}
```

8. Barcode SDK API Calls

8.1 Implement *ISbtSdkApiDelegate* protocol

Objective C protocol which defines SDK callbacks interface. Registration of a particular object which conforms to *ISbtSdkApiDelegate* protocol is required to receive particular from the SDK. SDK callback interface is defined by *ISbtSdkApiDelegate* Objective C protocol. Registration of a particular object which conforms to *ISbtSdkApiDelegate* protocol is required to receive particular notifications from Zebra Bluetooth Scanner iOS SDK.

BarcodeViewController.h file.

```
#import <UIKit/UIKit.h>
#import "SbtSdkFactory.h"

/// Responsible for barcode sdk events and action
@interface BarcodeViewController : UIViewController<ISbtSdkApiDelegate> {



}

@end
```

BarcodeViewController.m file.

```
#import "BarcodeViewController.h"

/// Responsible for barcode sdk events and action
@interface BarcodeViewController ()

@end

@implementation BarcodeViewController

#pragma mark - Life Cycle Methods
```

```
- (void)viewDidLoad {

    [super viewDidLoad];
    [sdkApi sbtSetDelegate:self];

}
/// The barcode event
/// @param barcodeData The barcode data
/// @param barcodeType The barcode type
/// @param scannerID  The scanner id
- (void)sbtEventBarcode:(NSString *)barcodeData barcodeType:(int)barcodeType
fromScanner:(int)scannerID {

    NSLog(@"Barcode Event: data event, %@",barcodeData);

}

/// The barcode event data
/// @param barcodeData The barcode data
/// @param barcodeType The barcode type
/// @param scannerID  The scanner id
- (void)sbtEventBarcodeData:(NSData *)barcodeData barcodeType:(int)barcodeType
fromScanner:(int)scannerID {

    NSData *decodeData = [[NSData alloc] initWithData:barcodeData];
    NSString *decodeDataString = [[NSString alloc] initWithBytes:((unsigned char*)[decodeData bytes])
length:([decodeData length]) encoding:NSUTF8StringEncoding];

    NSLog(@"Barcode Event :  %@",decodeDataString);
    dispatch_async(dispatch_get_main_queue(),^{
        self->textView_barcode_data.text = decodeDataString;
    });
```

```
}

/// The device connected event
/// @param activeScanner The connected scanner object
- (void)sbtEventCommunicationSessionEstablished:(SbtScannerInfo *)activeScanner {

    NSLog(@"Device has connected, Device name  :%@",[activeScanner getScannerName]);

}

/// The device disconnected event
/// @param scannerID The scanner id
- (void)sbtEventCommunicationSessionTerminated:(int)scannerID {

    NSLog(@"Device has Diconnected, Device ID %d",scannerID);

}

/// The firmware update event
/// @param fwUpdateEventObj firmware update event object
- (void)sbtEventFirmwareUpdate:(FirmwareUpdateEvent *)fwUpdateEventObj {

    NSLog(@"Firmware updat event - Max record : %d",fwUpdateEventObj.maxRecords);
    NSLog(@"Firmware updat event - Current record : %d",fwUpdateEventObj.currentRecord);
    NSLog(@"Firmware updat event - Current Status : %d",fwUpdateEventObj.status);
}


/// The image event
/// @param imageData The image data
/// @param scannerID The scanner id
- (void)sbtEventImage:(NSData *)imageData fromScanner:(int)scannerID {
```

```
    NSLog(@"Image event");

}

/// The device appear event
/// @param availableScanner The scanner object
- (void)sbtEventScannerAppeared:(SbtScannerInfo *)availableScanner {

    NSLog(@"Device has appeared, Device name %@",[availableScanner getScannerName]);

}

/// The scanner disappear event
/// @param scannerID The scanner id
- (void)sbtEventScannerDisappeared:(int)scannerID {

    NSLog(@"Device disappeared");
}



/// The video event
/// @param videoFrame The video data
/// @param scannerID The scanner id
- (void)sbtEventVideo:(NSData *)videoFrame fromScanner:(int)scannerID {

    NSLog(@"The video event");

}

@end
```

8.2 Initialize barcode sdk

```
/// Initilize barcode sdk
-(void)initilizeBarcodeSDK {

    sdkApi = [SbtSdkFactory createSbtSdkApiInstance];
    [sdkApi sbtSetDelegate:self];
    [sdkApi sbtSetOperationalMode:SBT_OPMODE_ALL];
    [sdkApi sbtSubsribeForEvents:SBT_EVENT_SCANNER_APPEARANCE |
    SBT_EVENT_SCANNER_DISAPPEARANCE | SBT_EVENT_SESSION_ESTABLISHMENT |
    SBT_EVENT_SESSION_TERMINATION | SBT_EVENT_BARCODE | SBT_EVENT_IMAGE |
    SBT_EVENT_VIDEO];
    [sdkApi sbtEnableAvailableScannersDetection:YES];


}
```

8.3 Get barcode sdk version

Returns version of the SDK.

```
/*
This method will provide the scanner SDK version
- Returns : SDK version
*/
- (NSString *)getSDKVersion
{
    NSString *version = [sdkApi sbtGetVersion];
    return version;

}
```

8.4 Connect

Requests to establish communication session with a particular available scanner in “SSI” mode.

```
/// This method will initiate the connection with particuler scanner
/// @param scanner_id Scanner id of the connecting scanner
-(void)connectScanner:(int)scanner_id{
    if (sdkApi != nil)
    {
      if(scanner_id != -1 )
      {
        SBT_RESULT conn_result = [sdkApi sbtEstablishCommunicationSession:scanner_id];

        if (SBT_RESULT_SUCCESS != conn_result)
        {
            dispatch_async(dispatch_get_main_queue(), ^{
                [self showMessageBox:@"SCANNER_CONNECTION_FAILED"];
            });
        }
      }
    }
}
```

8.5 Disconnect

Requests to terminate communication session with a particular active scanner.

```
/// This method will initiate the disconnection with particuler scanner
/// @param scannerId Scanner id of the disconnecting the scanner
- (void)disconnect:(int)scannerId
{

    if (sdkApi != nil)
```

```
    {
        SBT_RESULT res = [sdkApi sbtTerminateCommunicationSession:scannerId];
        if (res == SBT_RESULT_FAILURE) {
            [self showMessageBox:@"DISCONNECT_FAILED_MESSAGE"];
        }
    }
}
```

9. Firmware Update

9.1 Overview

To do a firmware update in the 123RFID mobile app, you needed a firmware file in ".dat"/.SCNPLG" format.

9.2 Implement Firmware update

```
- (IBAction)btnFirmwaareUpdate:(id)sender
{
    NSString *inputXML = [NSString stringWithFormat:@"<inArgs><scannerID>%d</scannerID><cmdArgs><arg-string>%@</arg-string></cmdArgs></inArgs>", _connectedRederID, @"FIRMWARE_FILE_PATH"];
    int firmwareFileTypeCommand = 0;

    //If firmware file is ".Dat" then command type is "SBT_UPDATE_FIRMWARE".
    //If firmware file is plugin then command type is "SBT_UPDATE_FIRMWARE_FROM_PLUGIN".
    firmwareFileTypeCommand = SBT_UPDATE_FIRMWARE;
    SBT_RESULT result = [self executeCommand:firmwareFileTypeCommand aInXML:inputXML];

    if (result) {

        NSString *in_xml = [NSString stringWithFormat:@"<inArgs><scannerID>%d</scannerID></inArgs>", _connectedRederID];
        [self performStartNewFirmware:in_xml];

    }else{
        NSLog(@"Firmare update failed !");
    }

}

/// Perform start new firmware
/// @param param The inXML value
- (void)performStartNewFirmware:(NSString*)param
```

```
{

    SBT_RESULT result = [self executeCommand:SBT_START_NEW_FIRMWARE aInXML:param aOutXML:nil
forScanner:_connectedRederID];
    if (result != SBT_RESULT_SUCCESS)
    {
        NSLog(@"Firmware Update Failed.");

    }else{
        NSLog(@"Firmware Update Success.");
    }


}


// Firmware update event
/// @param fwUpdateEventObj SDK's firmware update event object
- (void)sbtEventFirmwareUpdate:(FirmwareUpdateEvent *)event{

     NSLog(@"Current Record : %f",(float)event.currentRecord);
     NSLog(@"Max Record : %f",(float)event.maxRecords);
    int currentProgressInPrecentage = (int)((float)event.currentRecord/event.maxRecords*100);
    NSLog(@"Precentage %d",currentProgressInPrecentage);


}


/// Execute command inXML only
/// @param opCode Command code
/// @param inXML Input XML
/// @Return SBT Result
- (SBT_RESULT)executeCommand:(int)opCode aInXML:(NSString*)inXML{
    if (sdkApi != nil){

        SBT_RESULT resultExecuteCommand = [sdkApi sbtExecuteCommand:opCode aInXML:inXML aOutXML:NULL
forScanner:_connectedRederID];
        return resultExecuteCommand;
```

```
    }
    return SBT_RESULT_FAILURE;

}
```

10. Locate Reader

The SDK supports performing locate the reader by calling "srfidLocateReader". After calling this API reader will beep.

The following example demonstrates perform locate reader.

```
- (void) locateTheReader:(BOOL)enabled
{
    SRFID_RESULT conn_result = SRFID_RESULT_FAILURE;
    if (self->_apiInstance != nil)
    {
        conn_result = [self->_apiInstance srfidLocateReader:_connectedRederID doEnabled:enabled aStatusMessage:nil];

        if (SRFID_RESULT_SUCCESS != conn_result)
        {
            NSLog(@"Couldn't locate reader");
        } else{
            NSLog(@"Locate the reader");
        }
    }

}
```

11. Batch Mode

11.1 Get Batch Mode

This "srfidGetBatchModeConfig" API will return the status ("BATCHMODECONFIG") of the batch mode.

```
typedef enum
{

   SRFID_BATCHMODECONFIG_DISABLE  = 0x00,
   SRFID_BATCHMODECONFIG_AUTO     = 0x01,
   SRFID_BATCHMODECONFIG_ENABLE   = 0x02,

} SRFID_BATCHMODECONFIG;



-(SRFID_BATCHMODECONFIG)getBatchModeConfig:(NSString **)responsMessage
{
   //SRFID_BATCHMODECONFIG_AUTO ,SRFID_BATCHMODECONFIG_ENABLE and
SRFID_BATCHMODECONFIG_DISABLE
   SRFID_BATCHMODECONFIG batchModeConfigiuration = SRFID_BATCHMODECONFIG_AUTO;
   SRFID_RESULT srfid_result = SRFID_RESULT_FAILURE;

   for(int i = 0; i < 2; i++)
   {
      srfid_result = [self->_apiInstance srfidGetBatchModeConfig:_connectedRederID
aBatchModeConfig:&batchModeConfigiuration aStatusMessage:responsMessage ];
      if ((srfid_result != SRFID_RESULT_RESPONSE_TIMEOUT) && (srfid_result != SRFID_RESULT_FAILURE)) {
         break;
      }
   }

   if (srfid_result == SRFID_RESULT_SUCCESS)
   {
```

```
        NSLog(@"getBatchMod Response Success: %u", batchModeConfigiuration);
    }
    else if(srfid_result == SRFID_RESULT_RESPONSE_ERROR)
    {
        NSLog(@"Response Error");
    }
    else if(srfid_result == SRFID_RESULT_FAILURE || srfid_result == SRFID_RESULT_RESPONSE_TIMEOUT)
    {
        NSLog(@"Timeout or Failure");
    }

    return batchModeConfigiuration;
}




- (IBAction)btnGetBatchMode:(id)sender
{
    SRFID_BATCHMODECONFIG batchModeConfigiuration =  [self getBatchModeConfig:nil];

    switch (batchModeConfigiuration) {
        case SRFID_BATCHMODECONFIG_DISABLE:
            NSLog(@"Batchmode Disable");
            break;
        case SRFID_BATCHMODECONFIG_AUTO:
            NSLog(@"Batchmode Auto");
            break;
        case SRFID_BATCHMODECONFIG_ENABLE:
            NSLog(@"Batchmode Enable");
            break;
        default:
            break;
    }
```

```
}
```

11.2 Set Batch Mode

This "srfidSetBatchModeConfig" API will set the batch mode.

```
- (void)setBatchModeConfig:(NSString **)statusMessage
aBatchModeConfig:(SRFID_BATCHMODECONFIG)batchModeConfig
{
   //SRFID_BATCHMODECONFIG_AUTO ,SRFID_BATCHMODECONFIG_ENABLE and
SRFID_BATCHMODECONFIG_DISABLE
   SRFID_RESULT srfid_result = SRFID_RESULT_FAILURE;

   for(int i = 0; i < 2; i++)
   {
      srfid_result = [self->_apiInstance srfidSetBatchModeConfig:_connectedRederID
aBatchModeConfig:batchModeConfig aStatusMessage:statusMessage];
      if ((srfid_result != SRFID_RESULT_RESPONSE_TIMEOUT) && (srfid_result != SRFID_RESULT_FAILURE)) {
         break;
      }
   }

   if (srfid_result == SRFID_RESULT_SUCCESS)
   {
     NSLog(@"setBatchModeConfig Response Success: %u", srfid_result);

   }
   else if(srfid_result == SRFID_RESULT_RESPONSE_ERROR)
   {
      NSLog(@"Response Error");
   }
   else if(srfid_result == SRFID_RESULT_FAILURE || srfid_result == SRFID_RESULT_RESPONSE_TIMEOUT)
   {
      NSLog(@"Timeout or Failure");
   }
```

```
}
```

11.3 Get Tags in Batch Mode

This "srfidgetTags" API will request to receive tags read in batch mode from a particular RFID reader.

```
- (SRFID_RESULT)getTags:(NSString **)statusMessage
{

    NSString *status_msg = nil;

    if (nil != self->_apiInstance)
    {
        SRFID_RESULT result;
        result = [self->_apiInstance srfidgetTags:_connectedRederID aStatusMessage:&status_msg];
        return result;
    }
    return SRFID_RESULT_FAILURE;

}
```

11.4 Purge Tag

Request to purge tags read in batch mode from a particular RFID reader.

```
- (SRFID_RESULT)purgeTags:(NSString **)statusMessage
{
    NSString *status_msg = nil;

    if (nil != self->_apiInstance)
    {
        SRFID_RESULT result;
        result = [self->_apiInstance srfidPurgeTags:self->_connectedRederID aStatusMessage:&status_msg];
        return result;
    }
    return SRFID_RESULT_FAILURE;

}
```

11.5 Get Reader Configuration

Request to get the reader configurations after batch mode reconnect.

```
- (void) reconnectAfterBatchMode
{
   [self->_apiInstance srfidGetConfigurations];

}
```

12. Auto Reconnect

Requests to enable/disable "Automatic communication session reestablishment" option.

```
[apiInstance srfidEnableAutomaticSessionReestablishment:YES];
```

```
- (SRFID_RESULT) srfidEnableAutomaticSessionReestablishment:(BOOL)enable;
```

Parameters

(BOOL)enable

[in] Whether the option should be enabled or disabled:

YES

Requests to enable "Automatic communication session reestablishment" option.

NO

Requests to disable "Automatic communication session reestablishment" option.

Return Values

SRFID_RESULT_SUCCESS

"Automatic communication session reestablishment" option has been enabled/disabled successfully.

Notes

If the option is enabled the SDK will automatically establish communication session with the last active RFID reader that had unexpectedly disappeared once the RFID reader will be recognized as available:

The RFID reader could be recognized as available automatically by SDK if "Available readers detection" option is enabled.

The RFID reader could be recognized as available during discovery procedure requested by srfidGetAvailableReadersList API.

"Session Established" notification will be provided once the communication session is established, if this type of notification is enabled.

13. Access Sequence

This API is used to execute multiple access operations (Read, Write, etc) at the same time.

```
- (SRFID_RESULT) srfidPerformAccessInSequence:(int)readerID aAccessCriteria:(srfidAccessCriteria*)accessCriteria
aAccessParameters: (NSArray *)accessParameters aStatusMessage:(NSString**)statusMessage;
```

Parameters

(int)readerID

[in] Unique identifier of a particular RFID reader assigned by SDK.

(srfidAccessCriteria*)accessCriteria

[in] Access criteria to identify the Tag on which the block erase operation needs to be carried out by the SDK. Using the Access Criteria a tag can be chosen with one of the memory bank data.

(NSArray)accessParameters

[Array]accessParameters is to identify the list of accesses (Read,Write,Lock,Kill) shall be performed, each array object is of type RfidAccessParameters .

(NSString**)statusMessage

[out] Pointer to NSString variable intended for storage of status message if an error has been reported by the RFID reader via ASCII interface.

Return Values

SRFID_RESULT_SUCCESS

Block erase operation has been started successfully.

SRFID_RESULT_FAILURE

SDK has failed to perform block erase operation.

SRFID_READER_NOT_AVAILABLE

The request was not processed because the RFID reader specified by readerID parameter was not active or available.

SRFID_RESULT_INVALID_PARAMS

Invalid parameters (e.g. an identifier of memory bank is not specified).

SRFID_RESULT_RESPONSE_ERROR

An error has been reported by the RFID reader via ASCII interface.

SRFID_RESULT_RESPONSE_TIMEOUT

Timeout has occurred while waiting for a response from the RFID reader.

Notes

- If an error has been reported by the RFID reader the received error message is stored in statusMessage parameter.

Create access params for write

```
-(srfidAccessParameters*)setAccessParamsForWrite:(SRFID_ACCESSOPERATIONCODE)opCode
memoryBank:(SRFID_MEMORYBANK)memoryBank offset:(int)offset
 password:(int)password doBlockWrite:(BOOL)doBlockWrite dataToWrite:(NSString*)dataToWrite {

    srfidAccessParameters *accesParams = [[srfidAccessParameters alloc] init];
    accesParams.accessOperationCode = opCode;
    accesParams.memoryBank = memoryBank;
    accesParams.offset = offset;
    accesParams.password = password;
    accesParams.doBlockWrite = doBlockWrite;
    accesParams.dataToWrite = dataToWrite;
    return accesParams;


}
```

Create access params for Lock

```
-(srfidAccessParameters*)setAccessParamsForLock:(SRFID_ACCESSOPERATIONCODE)opCode
memoryBank:(SRFID_MEMORYBANK)memoryBank password:(int)password
accPermission:(SRFID_ACCESSPERMISSION)accPermission
{
    srfidAccessParameters *accesParams = [[srfidAccessParameters alloc] init];
    accesParams.accessOperationCode = opCode;
    accesParams.memoryBank = memoryBank;
    accesParams.accessPermissions = accPermission;
    return accesParams;

}
```

Create access params for Read

```
-(srfidAccessParameters*)setAccessCriteriaPramForRead:(SRFID_ACCESSOPERATIONCODE)opCode
memoryBank:(SRFID_MEMORYBANK)memoryBank offset:(int)offset length:(int)length password:(int)password {

    srfidAccessParameters *accesParams = [[srfidAccessParameters alloc] init];
    accesParams.accessOperationCode = opCode;
```

```
    accesParams.memoryBank = memoryBank;
    accesParams.offset = offset;
    accesParams.length = length;
    accesParams.password = password;
    return accesParams;
}
```

```
-(void)accessSequence{

    // initialize access criteria
    srfidAccessCriteria *accessCriteria = [[srfidAccessCriteria alloc] init];
    // setup tag filter 1
    srfidTagFilter *tagFilter1 = [[srfidTagFilter alloc] init];
    [tagFilter1 setFilterMaskBank:SRFID_MEMORYBANK_EPC];
    [tagFilter1 setFilterData:@"E280689400004006 5071E164"];
    [tagFilter1 setFilterDoMatch:YES];
    [tagFilter1 setFilterMask:@"FFFFFFFF"];
    [tagFilter1 setFilterMaskStartPos:2];
    [tagFilter1 setFilterMatchLength:2];
    [accessCriteria setTagFilter1:tagFilter1];

    NSMutableArray *accessParamsArray = [[NSMutableArray alloc]init];
    [accessParamsArray addObject:[self setAccessCriteriaPramForRead:SRFID_ACCESSOPERATIONCODE_READ
memoryBank:SRFID_MEMORYBANK_EPC offset:0 length:8 password:0]];
    [accessParamsArray addObject:[self setAccessCriteriaPramForRead:SRFID_ACCESSOPERATIONCODE_READ
memoryBank:SRFID_MEMORYBANK_TID offset:0 length:2 password:0]];
```

```
    [accessParamsArray addObject:[self setAccessCriteriaPramForRead:SRFID_ACCESSOPERATIONCODE_READ
memoryBank:SRFID_MEMORYBANK_USER offset:0 length:0 password:0]];
    [accessParamsArray addObject:[self setAccessCriteriaPramForRead:SRFID_ACCESSOPERATIONCODE_READ
memoryBank:SRFID_MEMORYBANK_RESV offset:0 length:4 password:0]];
    [accessParamsArray addObject:[self setAccessParamsForWrite:SRFID_ACCESSOPERATIONCODE_WRITE
memoryBank:SRFID_MEMORYBANK_EPC offset:3  password:00 doBlockWrite:false
dataToWrite:@"1111222233334444555"]];

    SRFID_RESULT result;
    result = [self->_apiInstance srfidPerformAccessInSequence:self->_connectedRederID aAccessCriteria:accessCriteria
aAccessParameters:accessParamsArray aStatusMessage:nil];
    NSLog(@"Result Perform Access In Sequence %u",result);

}
```

14. Set Attributes

Reader off mode timeout parameter: byte parameter number **1765**, default 30 minutes (default value 29 = 0x1D, which means 30 minutes).

```
#pragma mark - Methods - Set Offline timeout attribute

-(void)setAttributes {

    srfidAttribute *attributeDet =[[srfidAttribute alloc]init];
    [attributeDet setAttrNum:1765];
    [attributeDet setAttrType:@"B"];
    [attributeDet setAttrVal:[NSString stringWithFormat:@"10"]];

    /* cause RFID reader to generate asynchronous battery status notification */
    SRFID_RESULT result = [apiInstance srfidSetAttribute:connectedReaderId aAttrInfo:attributeDet
aStatusMessage:nil];

    if (SRFID_RESULT_SUCCESS == result) {
        NSLog(@"Set attributes Success");
    }
    else {
        NSLog(@"Failed to set attribute");
    }

}
```

15. Access Sequence

This API is used to execute multiple access operations (Read, Write, etc) at the same time.

```
- (SRFID_RESULT) srfidPerformAccessInSequence:(int)readerID aAccessCriteria:(srfidAccessCriteria*)accessCriteria
aAccessParameters: (NSArray *)accessParameters aStatusMessage:(NSString**)statusMessage;
```

Parameters

(int)readerID

[in] Unique identifier of a particular RFID reader assigned by SDK.

(srfidAccessCriteria*)accessCriteria

[in] Access criteria to identify the Tag on which the block erase operation needs to be carried out by the SDK. Using the Access Criteria a tag can be chosen with one of the memory bank data.

(NSArray)accessParameters

[Array]accessParameters is to identify the list of accesses (Read,Write,Lock,Kill) shall be performed, each array object is of type RfidAccessParameters .

(NSString**)statusMessage

[out] Pointer to NSString variable intended for storage of status message if an error has been reported by the RFID reader via ASCII interface.

Return Values

SRFID_RESULT_SUCCESS

Block erase operation has been started successfully.

SRFID_RESULT_FAILURE

SDK has failed to perform block erase operation.

SRFID_READER_NOT_AVAILABLE

The request was not processed because the RFID reader specified by readerID parameter was not active or available.

SRFID_RESULT_INVALID_PARAMS

Invalid parameters (e.g. an identifier of memory bank is not specified).

SRFID_RESULT_RESPONSE_ERROR

An error has been reported by the RFID reader via ASCII interface.

SRFID_RESULT_RESPONSE_TIMEOUT

Timeout has occurred while waiting for a response from the RFID reader.

Notes

- If an error has been reported by the RFID reader the received error message is stored in statusMessage parameter.

Create access params for write

```
-(srfidAccessParameters*)setAccessParamsForWrite:(SRFID_ACCESSOPERATIONCODE)opCode
memoryBank:(SRFID_MEMORYBANK)memoryBank offset:(int)offset
 password:(int)password doBlockWrite:(BOOL)doBlockWrite dataToWrite:(NSString*)dataToWrite {

    srfidAccessParameters *accesParams = [[srfidAccessParameters alloc] init];
    accesParams.accessOperationCode = opCode;
    accesParams.memoryBank = memoryBank;
    accesParams.offset = offset;
    accesParams.password = password;
    accesParams.doBlockWrite = doBlockWrite;
    accesParams.dataToWrite = dataToWrite;
    return accesParams;

}
```

Create access params for Lock

```
-(srfidAccessParameters*)setAccessParamsForLock:(SRFID_ACCESSOPERATIONCODE)opCode
memoryBank:(SRFID_MEMORYBANK)memoryBank password:(int)password
accPermission:(SRFID_ACCESSPERMISSION)accPermission
{
    srfidAccessParameters *accesParams = [[srfidAccessParameters alloc] init];
    accesParams.accessOperationCode = opCode;
    accesParams.memoryBank = memoryBank;
    accesParams.accessPermissions = accPermission;
    return accesParams;
}
```

Create access params for Read

```
-(srfidAccessParameters*)setAccessCriteriaPramForRead:(SRFID_ACCESSOPERATIONCODE)opCode
memoryBank:(SRFID_MEMORYBANK)memoryBank offset:(int)offset length:(int)length password:(int)password {
```

```
    srfidAccessParameters *accesParams = [[srfidAccessParameters alloc] init];
    accesParams.accessOperationCode = opCode;
    accesParams.memoryBank = memoryBank;
    accesParams.offset = offset;
    accesParams.length = length;
    accesParams.password = password;
    return accesParams;
}
```

```
-(void)accessSequence{

    // initialize access criteria
    srfidAccessCriteria *accessCriteria = [[srfidAccessCriteria alloc] init];
    // setup tag filter 1
    srfidTagFilter *tagFilter1 = [[srfidTagFilter alloc] init];
    [tagFilter1 setFilterMaskBank:SRFID_MEMORYBANK_EPC];
    [tagFilter1 setFilterData:@"E2806894000040065071E164"];
    [tagFilter1 setFilterDoMatch:YES];
    [tagFilter1 setFilterMask:@"FFFFFFFF"];
    [tagFilter1 setFilterMaskStartPos:2];
    [tagFilter1 setFilterMatchLength:2];
    [accessCriteria setTagFilter1:tagFilter1];

    NSMutableArray *accessParamsArray = [[NSMutableArray alloc]init];
    [accessParamsArray addObject:[self setAccessCriteriaPramForRead:SRFID_ACCESSOPERATIONCODE_READ
memoryBank:SRFID_MEMORYBANK_EPC offset:0 length:8 password:0]];
```

```
    [accessParamsArray addObject:[self setAccessCriteriaPramForRead:SRFID_ACCESSOPERATIONCODE_READ
memoryBank:SRFID_MEMORYBANK_TID offset:0 length:2 password:0]];
    [accessParamsArray addObject:[self setAccessCriteriaPramForRead:SRFID_ACCESSOPERATIONCODE_READ
memoryBank:SRFID_MEMORYBANK_USER offset:0 length:0 password:0]];
    [accessParamsArray addObject:[self setAccessCriteriaPramForRead:SRFID_ACCESSOPERATIONCODE_READ
memoryBank:SRFID_MEMORYBANK_RESV offset:0 length:4 password:0]];
    [accessParamsArray addObject:[self setAccessParamsForWrite:SRFID_ACCESSOPERATIONCODE_WRITE
memoryBank:SRFID_MEMORYBANK_EPC offset:3  password:00 doBlockWrite:false
dataToWrite:@"1111222233344445555"]];

    SRFID_RESULT result;
    result = [self->_apiInstance srfidPerformAccessInSequence:self->_connectedRederID aAccessCriteria:accessCriteria
aAccessParameters:accessParamsArray aStatusMessage:nil];
    NSLog(@"Result Perform Access In Sequence %u",result);


}
```

16. Trigger Key Remapping

16.1 Set Trigger Key Configuration

This "srfidSetTriggerConfig" API will set the trigger key.

```
- (SRFID_RESULT)setTriggerConfigurationUpperTrigger:(SRFID_NEW_ENUM_KEYLAYOUT_TYPE)upper
andLowerTrigger:(SRFID_NEW_ENUM_KEYLAYOUT_TYPE)lower{

    SRFID_NEW_ENUM_KEYLAYOUT_TYPE upperTrigger = upper;
    SRFID_NEW_ENUM_KEYLAYOUT_TYPE lowerTrigger = lower;

    SRFID_RESULT srfid_result = SRFID_RESULT_FAILURE;

    for(int i = 0; i < 2; i++)
    {
        srfid_result = [_apiInstance srfidSetKeylayoutType: connectedRederID upperTrigger:upperTrigger
lowerTrigger:lowerTrigger];

        if ((srfid_result != SRFID_RESULT_RESPONSE_TIMEOUT) && (srfid_result != SRFID_RESULT_FAILURE)) {
            break;
        }
    }

    if (srfid_result == SRFID_RESULT_SUCCESS)
    {
        NSLog(@"Set Trigger Config Sucess ");
    }
    else if(srfid_result == SRFID_RESULT_RESPONSE_ERROR)
    {
        NSLog(@"Set Trigger Config Error ");
    }
    else if(srfid_result == SRFID_RESULT_FAILURE )
    {
        NSLog(@"Set Trigger Config Fail ");
    }
    else if(srfid_result == SRFID_RESULT_RESPONSE_TIMEOUT)
```

```
    {
        NSLog(@"Set Trigger Config Time out ");
    }

    return srfid_result;
}
```

16.2 Get Trigger Key Configuration

This "srfidGetTriggerConfig" API will get the trigger key configuration.

```
- (SRFID_RESULT)getTriggerConfiguration{

    SRFID_NEW_ENUM_KEYLAYOUT_TYPE upperTrigger = RFID_SCAN;
    SRFID_NEW_ENUM_KEYLAYOUT_TYPE lowerTrigger = TERMINAL_SCAN;

    SRFID_RESULT srfid_result = SRFID_RESULT_FAILURE;

    for(int i = 0; i < 2; i++)
    {
            srfid_result = [_apiInstance srfidGetKeylayoutType: connectedRederID upperTrigger: &upperTrigger
lowerTrigger: &lowerTrigger];

        if ((srfid_result != SRFID_RESULT_RESPONSE_TIMEOUT) && (srfid_result != SRFID_RESULT_FAILURE)) {
            break;
        }
    }

    if (srfid_result == SRFID_RESULT_SUCCESS)
    {
```

```
        NSLog(@"Trigger Config Sucess upper trigger %u", upperTrigger);
            NSLog(@"Trigger Config Sucess lower trigger %u", lowerTrigger);
       [self getTriggerMapping:configuration];

    }
    else if(srfid_result == SRFID_RESULT_RESPONSE_ERROR)
    {
        NSLog(@"Trigger Config Error ");
    }
    else if(srfid_result == SRFID_RESULT_FAILURE)
    {
        NSLog(@"Trigger Config Failled ");
    }
    else if(srfid_result == SRFID_RESULT_RESPONSE_TIMEOUT)
    {
        NSLog(@"Trigger Config Error ");
    }
    return srfid_result;
}
```

17. Factory Reset and Reboot

17.1 Factory Reset

Performing a factory reset will clear any saved settings and restart the reader. The region needs to be set again.

```
/// Factory reset the reader
/// @param readerID The reader id
/// @param statusMessage The status message

- (SRFID_RESULT)setReaderFactoryReset:(int)readerID status:(NSString **)statusMessage{

    SRFID_RESULT srfid_result = SRFID_RESULT_FAILURE;
    srfid_result = [m_RfidSdkApi srfidFactoryReset:readerID aStatusMessage:statusMessage];
    return srfid_result;

}
```

17.2 Reboot

The device will be rebooted.

```
/// Reboot the reader
/// @param readerID The reader id
/// @param statusMessage The status message

- (SRFID_RESULT)setReaderReboot:(int)readerID status:(NSString **)statusMessage{

    SRFID_RESULT srfid_result = SRFID_RESULT_FAILURE;
    srfid_result = [m_RfidSdkApi srfidReboot:readerID aStatusMessage:statusMessage];
    return srfid_result;

}
```

18. PP+ Battery Support

Performing a factory reset will clear any saved settings and restart the reader. The region needs to be set again.

```
/// Get battery status
/// @param readerID The reader id
/// @param statusMessage The status message
-(SRFID_RESULT)getBatteryStatus:(int)readerID  aStatusMessage:(NSString**)statusMessage {

    NSMutableArray *batteryStatusValueList = [[[NSMutableArray alloc] init] autorelease];
    SRFID_RESULT srfid_result = SRFID_RESULT_FAILURE;

    for(int i = 0; i < ZT_MAX_RETRY; i++)
    {
        srfid_result = [m_RfidSdkApi srfidGetBatteryStatus:[m_ActiveReader getReaderID]
batteryStatusArray:&batteryStatusValueList aStatusMessage:statusMessage];

        if ((srfid_result != SRFID_RESULT_RESPONSE_TIMEOUT) && (srfid_result != SRFID_RESULT_FAILURE))
        {
            break;
        }
    }

    if (srfid_result == SRFID_RESULT_SUCCESS)
    {

        [[[zt_RfidAppEngine sharedAppEngine] appConfiguration] setBatteryStatusArray:batteryStatusValueList];



    }
    else if(srfid_result == SRFID_RESULT_RESPONSE_ERROR)
    {
        // do nothing
    }
```

```
    else if(srfid_result == SRFID_RESULT_FAILURE || srfid_result == SRFID_RESULT_RESPONSE_TIMEOUT)
    {
        [self readerProblem];
    }
    return srfid_result;


}
```

19. Async Tag Read/Write

19.1 Async Tag Read

Read tag asynchronous. This method has following parameters.

- tagID: Selected tag ID.
- tagData: TagData object.
- memoryBankID: Selected memory bank.
- offset: Offset for the write operation.
- data – Selected tag data.
- password: Password for the write operation.
- statusMessage: Status message to return.

```
SRFID_RESULT readTagAsync:(NSString*)tagID withTagData:(srfidTagData **)tagData
withMemoryBankID:(SRFID_MEMORYBANK)memoryBankID withOffset:(short)offset withLength:(short)length
withPassword:(long)password aStatusMessage:(NSString**)statusMessage
{
  if (_apiInstance != nil)
  {

    return [_apiInstance srfidReadTagAsync:[m_ActiveReader getReaderID] aAccessCriteria:(srfidAccessCriteria*)tagID
aMemoryBank:memoryBankID aOffset:offset aLength:length aPassword:password aStatusMessage:statusMessage];
  }

  return SRFID_RESULT_FAILURE;

}
```

Inside the success method, we should call the abort operation when the async read method is getting called.

```
SRFID_RESULT rfid_res = SRFID_RESULT_FAILURE;
```

```
rfid_res = [[[zt_RfidAppEngine sharedAppEngine] operationEngine] stopInventory:nil];
```

19.2 Async Tag Write

Write tag asynchronous. This method has following parameters.

- tagID: Selected tag ID.
- tagData: TagData object.
- memoryBankID: Selected memory bank.
- offset: Offset for the write operation.
- data – Selected tag data.
- password: Password for the write operation.
- blockWrite: Block write access for write operation.
- statusMessage: Status message to return.

```
   SRFID_RESULT writeTagAsync:(NSString*)tagID withTagData:(srfidTagData **)tagData
withMemoryBankID:(SRFID_MEMORYBANK)memoryBankID withOffset:(short)offset withLength:(short)length
withPassword:(long)password doBlockWrite:(BOOL)blockwrite aStatusMessage:(NSString**)statusMessage
{
   if (_apiInstance != nil)
   {

      return [_apiInstance srfidReadTagAsync:[m_ActiveReader getReaderID] aAccessCriteria:(srfidAccessCriteria*)tagID
aMemoryBank:memoryBankID aOffset:offset aLength:length aPassword:password aDoBlockWrite:blockWrite
aStatusMessage:statusMessage];
   }

   return SRFID_RESULT_FAILURE;

}
```

20. Scanner Batch Mode

Scanner batch mode allows you to scan barcodes without connecting to the mobile app, and once you connected to the mobile application and navigate to the barcode tab, the scanned barcodes will be there.

```
/// Batch request

/// @Return SBT Result

-(SBT_RESULT)scanBatchRequest{

    SbtScannerInfo *scannerInfo = [[ScannerEngine sharedScannerEngine] getConnectedScannerInfo];

    if (scannerInfo != NULL){

        NSString *inXML = [NSString stringWithFormat:SCANNER_PULL_RELEASE_TRIGGER_SCAN_XML,[scannerInfo
getScannerID]];



        return [[ScannerEngine sharedScannerEngine] executeCommand:SBT_DEVICE_BATCH_REQUEST aInXML:inXML];

    }

    return SBT_RESULT_FAILURE;

}
```

**Following table contains the constant values to the above variables. (You can directly replace those variable names with the corresponding values)

Variable	Value
SCANNER_PULL_RELEASE_TRIGGER_SCAN_XML	@"<inArgs><scannerID>%d</scannerID></inArgs>"
SBT_DEVICE_BATCH_REQUEST	0x7DF
SBT_RESULT_FAILURE	0x01