React Native (v2.7.x)

Integrate the cmbSDK React-Native Module in your App

Download the plugin via npm

```
$ npm install cmbsdk-react-native --save
```

Mostly automatic installation

```
$ react-native link cmbsdk-react-native
```

Import the component in your react-native app by adding this

- JavaScript

```
import { CMBReader, cmb } from 'cmbsdk-react-native';
```

You can access all of the API methods through the **cmb** constant, and all of the enums and constants are available in the CMBReader class.

Before continuing, download the cmbSDK React-Native zip file from our download section, and see the API reference.

Download page: https://cmbdn.cognex.com/downloads/cmbSDK

API reference: https://cmbdn.cognex.com/knowledge/react-nat/rct-api-metho

Integrating cmbSDK React-Native Module

Make sure you have downloaded the cmbSDK React-Native zip archive from our download page.

1. For iOS from the downloaded zip file, open the iOS directory.

   1.1 If you are using the mobile device's built in camera, do the same with the "MWBScannerImages.xcassets" file located in the iOS/Resources directory.

   1.2 In your projects info.plist file you need to add a key depending on the readerDevice type that you are using.

   - If you are using the device camera for scanning, add the "NSCameraUsageDescription" key with a description on how your app will use the camera (for example: Scanning barcodes").

   - If you're using a MX-1xxx device, you will need to add a "Supported external accessory protocols" array with an item "com.cognex.dmcc". You will also need to follow this GUIDE before publishing your MX-1xxx enabled app to the app store. This is ONLY required for scanning with an MX mobile terminal.

2. For Android there is no extra steps needed

And that's it. You should be able to run your app with the cmbSDK react-native module working.

API METHODS

loadScanner()

To get a scanner up and running, the first thing to do, is to call the **loadScanner()** method. It expects a CMBReader.DEVICE_TYPE param. This method does not connect to the Reader Device. We need to call **connect()** in the callback to actually connect to the Reader Device

```
cmb.loadScanner(CMBReader.DEVICE_TYPE.MXReader).then((response) => {
        cmb.getAvailability().then((response) => {
                if (response == CMBReader.AVAILABILITY.Available) {
                        connectReader();
                }
        }).catch((rejecter) => {
        })
});
```

connect()

```
/*   @return
    (promise)  {
        status : boolean, if connection succeeded true if not false
        err : string , if status false err will not be null
    }
*/
```

The result from the connect() method is returned as a Promise and it will contain the result of the connection attempt:

```
cmb.connect().then((connectMethodResult) => {
        // do something after a connection has been established
        configureReader();
}).catch((failure) => {
        console.log("CMB - connectReader failed: "+JSON.stringify(failure))
});
```

There is an Event Listener for the connection status of the ReaderDevice, namely the **CMBReader.EVENT.ConnectionStateChanged** event which is explained in more detail below.

disconnect()

```
/*   @return
    (promise)  {
        status : boolean, if disconnect succeeded true if not false
        err : string , if status false err will not be null
    }
*/
```

Just as there is **connect()**, there is a **disconnect()** method that does the opposite of **connect()** :

```
cmb.disconnect();
```

Similarly to **connect()**, **disconnect()** also triggers the **CMBReader.EVENT.ConnectionStateChanged** event.

startScanning()/stopScanning()

```
/*   @return Promise
      (bool) value of the Scanner Activity (true if the command was successful, false otherwise ex: if readerDevice not ini
*/
```

To start / stop the scanning process, we use these methods. They return a promise, which will be resolved if the command was successful (the scanning has started or stopped) or rejected otherwise (if there is no active ReaderDevice initialized or isn't connected).

After starting the scanner and scanning a barcode, the scan result triggers the **CMBReader.EVENT.ReadResultReceived** event.

scanImageFromUri()/scanImageFromBase64()

To scan a barcode from an image, we can use either scanImageFromUri or scanImageFromBase64.

scanImageFromUri Declaration

```
cmb.scanImageFromUri(uri).then((result) {
   ...
})
}).catch((error) => {
   ...
});
```

scanImageFromBase64 Declaration

```
cmb.scanImageFromBase64(base64).then((result) {
   ...
})
}).catch((error) => {
   ...
});
```

setSymbologyEnabled()

Once there is a connection to the Reader, we can enable symbologies by calling **setSymbologyEnabled()**. It expects three params: a CMBReader.SYMBOLOGY which is the symbology to be enabled or disabled, a boolean for ON/OFF, and a String for the commandID for handling the command result.

```
cmb.setSymbology(CMBReader.SYMBOLOGY.QR, true, CMBReader.SYMBOLOGY_NAME.QR);
```

This method triggers the **CMBReader.EVENT.CommandCompleted** event, whose result contains the *commandID* string from the third parameter, so that you know which commands have succeeded and which have failed in the event result.

isSymbologyEnabled()

To check if we have a symbol enabled, we use **isSymbologyEnabled()**. It takes two arguments: the CMBReader.SYMBOLOGY that we are checking, and a *commandID* string used for identifying the response. The result triggers the **CMBReader.EVENT.CommandCompleted** event, and it contains the *commandID* string from the second parameter, so that you know which commands have succeeded and which have failed in the event result.

```
cmb.isSymbologyEnabled(CMBReader.SYMBOLOGY.QR, CMBReader.SYMBOLOGY_NAME.QR);
```

setCameraMode()

This should be used only when using the device's built in camera for scanning (CMBReader.DEVICE_TYPE.Camera).

To set how the camera will behave when we use CAMERA device as a barcode reader, we use setCameraMode(). It takes a CMBReader.CAMERA_MODE argument.

```
cmb.setCameraMode(CMBReader.CAMERA_MODE)
/**
Use camera with no aimer. Preview is on, illumination is available.
CMBReader.CAMERA_MODE.NoAimer = 0,

Use camera with a basic aimer (e.g., StingRay). Preview is off, illumination is not available.
CMBReader.CAMERA_MODE.PassiveAimer = 1,

Use mobile device front camera. Preview is on, illumination is not available.
CMBReader.CAMERA_MODE.FrontCamera = 3
*/
```

*Note: CameraMode should be set BEFORE we call **loadScanner()** for it to take effect.*

enableCameraFlag/disableCameraFlag

Configure decoder flags for type specified by the code mask.

```
cmb.enableCameraFlag({"codeMask":codeMask, "flag":flag})
.then((response) => {
   ...
}).catch((rejecter) => {
   ...
});
```

```
cmb.disableCameraFlag({"codeMask":codeMask, "flag":flag})
.then((response) => {
   ...
}).catch((rejecter) => {
   ...
});
```

setPreviewOptions()

This should be used only when using the device's built in camera for scanning (CMBReader.DEVICE_TYPE.Camera).

This function expects one integer argument that is a result of the OR-ed result of all the preview options that we want enabled.

```
cmb.setPreviewOptions(CMBReader.CAMERA_PREVIEW_OPTION.NoZoomBtn | CMBReader.CAMERA_PREVIEW_OPTION.NoIllumBtn);
```

*Note: PreviewOptions should be set BEFORE we call **loadScanner()** for it to take effect.*

setPreviewContainerPositionAndSize()

This should be used only when using the device's built in camera for scanning (CMBReader.DEVICE_TYPE.Camera).

setPreviewContainerPositionAndSize takes an array argument with four integer objects, which are the X and Y values for the top left coordinate, and width and height values for the preview container size. All of the values are percentages of the device's screen.

```
cmb.setPreviewContainerPositionAndSize([0,0,100,50]);
//will set the preview to 0,0 and 100% width 50% height
```

setPreviewContainerFullScreen()

This should be used only when using the device's built in camera for scanning (CMBReader.DEVICE_TYPE.Camera).

Sets the camera preview to start in full screen instead of partial view.

```
cmb.setPreviewContainerFullScreen();
```

setPreviewContainerBelowStatusBar()

This should be used only when using the device's built in camera for scanning (CMBReader.DEVICE_TYPE.Camera).

Available only on iOS.

Sets the camera preview partial view top axis to start below the status bar matching the Androids behavior. It expects one boolean argument.

```
cmb.setPreviewContainerBelowStatusBar(true);
cmb.setPreviewContainerPositionAndSize([0,0,100,50]);
//will set the preview to 0,0 and 100% width 50% height.
//On iOS the partial view will be shown below the status bar.
```

```
cmb.setPreviewContainerBelowStatusBar(false);
cmb.setPreviewContainerPositionAndSize([0,0,100,50]);
//will set the preview to 0,0 and 100% width 50% height.
//On iOS the partial view will start from the top of the screen, and will overlap the status bar.
```

setPreviewOverlayMode()

Set the camera overlay mode. You need to do it before loadScanner is called, otherwise it will not work properly.

ONLY AVAILABLE ON MX-Mobile.

Read more about overlay modes here: iOS and Android

```
cmb.setPreviewOverlayMode(CMBReader.OVERLAY_MODE.CMB);
```

setStopScannerOnRotate()

Set whether or not the scanning session should stop when the application UI changes orientation. It expects one boolean argument.

```
cmb.setStopScannerOnRotate(true);
```

showToast()/hideToast()

To show or hide a message on top of the camera preview while scanning with the mobile camera.

```
cmb.showToast("Some message");
```

```
cmb.hideToast()
```

setLightsOn()

```
/*   @return
    (promise)  {
        status : boolean, true if successfully executed command
        err : string , if status false err will not be null
    }
*/
```

If we want to enable the flash we can use **setLightsOn()**. It expects one argument boolean and returns a promise.

isLightsOn()

```
/*   @return
    (promise)  {
```

```
        status : boolean, true if lights are on, false otherwise
        err : string , in case of error (e.g. reader not initialized)
    }
*/
```

We can check the lights status with isLightsOn(), which returns a promise.

enableImage()

Used to enable / disable image result type. Expects one boolean argument.

```
cmb.enableImage(true);
```

enableImageGraphics()

Used to enable / disable svg result type. Expects one boolean argument.

```
cmb.enableImageGraphics(true);
```

setParser()

Enable or disable parsing for scanned barcodes. Expects one argument of type CMBReader.RESULT_PARSER.

```
cmb.setParser(CMBReader.RESULT_PARSER.GS1);
```

setReadStringEncoding

Set encoding for the readString result type. Expects one argument of type CMBReader.READSTRING_ENCODING.

```
cmb.setReadStringEncoding(CMBReader.READSTRING_ENCODING.UTF_8);
```

sendCommand()

All the methods can be replaced with sending DMCC strings to the READER device. For that we can use our API method **sendCommand**. It can be used to control the Reader completely with command strings. It takes two string arguments, first of which is the DMCC itself, and the second one is to identify it in the response event.

More on the command strings can be found here or here.

```
cmb.sendCommand("SET SYMBOL.POSTNET OFF", "postnetCommandID");
```

beep()

Plays a beep on the reader device

beep()

setMDMReportingEnabled()

Available only on iOS.

A company owning and operating many Cognex Mobile Terminals may want to remotely collect up-to-date information about battery level, battery health, installed firmware, etc.

An iOS application using the cmbSDK framework can report status information of the attached Mobile Terminal to an MDM instance. This can be enabled with the setMDMReportingEnabled method that accepts one boolean argument.

More on the MDM Reporting can be found here

```
cmb.setMDMReportingEnabled(true);
```

createMDMAuthCredentials()

Available only on iOS.

Used for creating authentication credentials used for MDM reporting. It takes four string arguments: username, password, clientID and clientSecret.

Should be called before setMDMReportingEnabled.

More on the MDM Reporting can be found here

```
cmb.createMDMAuthCredentials("username", "password", "clientID", "clientSecret");
```

resetConfig()

```
/*   @return
    (promise)  {
        status : boolean, true if reset was successful, false otherwise
        err : string , in case of error (e.g. reader not initialized)
```

```
    }
*/
```

To reset the configuration options we can use **resetConfig.**

```
cmb.resetConfig(function(result){
  console.log(result);
})
```

registerSDK()

Another way to add your license key if you are using camera device. This one will overwrite your license key from manifest for Android or info.plist for iOS.

```
cmb.registerSDK("SDK_KEY");
```

getSdkVersion()

To see what cmbSDK version you're currently using:

```
cmb.getSdkVersion().then((response) => {
  console.log("Version: "+response);
});
```

getCameraExposureCompensationRange()

Get camera exposure compensation range.

Return successful callback with range as JSON object or error callback with the error message if something goes wrong.

```
cmb.getCameraExposureCompensationRange().then(function(range){
  // range is JSON object with these attributes
  // "lower" : min camera exposure value
  // "upper" : max camera exposure value
  // "step" : camera exposure step value
   console.log(JSON.stringify(range));
});
```

Note: The camera needs to be started within cmbSDK at least once to get the camera exposure compensation range.

setCameraExposureCompensation()

Sets the camera exposure compensation value. Send float value that will be set as exposure compensation.

```
cmb.setCameraExposureCompensation(5);
```

Note: This needs to be called after successful connection to reader device. If value that is send is greater than camera exposure max value, max value will be set, if value that is send is lower than camera exposure min value, min value will be set.

setCameraParam()

Configure decoder param id/value pair for type specified by the code mask.

```
/*   @return
     - success callback if param is set
     - error callback if something went wrong. The error object in this callback contains the error code and error message
*/
```

```
cmb.setCameraParam({"codeMask":codeMask, "param":param, "value":value})
.then((response) => {
   ...
}).catch((rejecter) => {
   ...
});
```

loadCameraConfig()

Load config from app data if exist.

```
/*   @return
     - success callback if config is loaded
     - error callback if something went wrong. The error object in this callback contains the error code and error message
*/
```

```
cmb.loadCameraConfig().then((response) => {
   ...
}).catch((rejecter) => {
   ...
});
```

Note: Use this API when reader device is used as camera only and it should be called when we have valid connection to reader. For handheld devices this API is not used and config is saved and automatically loaded from device memory.

setPairedBluetoothDevice

To set paired BT device that will be used for scanning. Input param for Android platform represent the **MAC address**, while for iOS input param represent the **UUID**.

For Android:

(void) ***setPairedBluetoothDevice(macAddress)***

For iOS:

(void) ***setPairedBluetoothDevice(uuid)***

Note: It should be called before we call **loadScanner()** *for it to take effect.*

getPairedBluetoothDevice()

To get paired BT device that is used for scanning. Returns current BT device UUID in successful callback.

(void) ***getPairedBluetoothDevice(successCallback, errorCallback)***

Note*: Method is only supported for iOS*

getAvailability

To check if reader device is available use getAvailability().

```
  cmb.getAvailability().then((response) => {
    if (response == CMBReader.AVAILABILITY.Available) {
        console.log('ReaderDevice is available');
    } else {
        console.log('ReaderDevice is NOT available');
    }
    }).catch((rejecter) => {
      console.log("CMB - getAvailability failed: "+JSON.stringify(rejecter))
  })
```

getConnectionState()

```
/**
@return A promise that resolves with the CMBReader.CONNECTION_STATE value of the current reader device
*/
```

If you need to get the current connection state, **getConnectionState()** can be used

```
cmb.getConnectionState().then(function(connectionState){
  if (connectionState == CMBReader.CONNECTION_STATE.Connected) {
              // reader is connected
```

```
  }
});
```

getDeviceBatteryLevel()

```
/*   @return
    (promise)  {
        value : int
        err : string , in case of error (e.g. reader not initialized)
    }
*/
```

Method to show the battery level of the connected device. Doesn't take any arguments.

Events

The React native cmbSDK module emits Events that can be used in the js application.

These should be added in the componentDidMount function, and removed in componentWillUnmount (see React component lifecycle).

First, create the event emitter:

```
import NativeModules.NativeEventEmitter;
const scannerListener = new NativeEventEmitter(cmb);
```

and then add listeners for each event you want to handle:

```
scannerListener.addListener(
      CMBReader.EVENT.ReadResultReceived,
      (result) => {
        if (result.goodRead == true){
                Alert.alert(
                  result.symbologyString,
                  result.readString
                );
        }
      }
    );
```

Here are all the events that the cmbSDK module can emit:

```
CMBReader.EVENT.ReadResultReceived
CMBReader.EVENT.AvailabilityChanged
CMBReader.EVENT.ConnectionStateChanged
CMBReader.EVENT.ScanningStateChanged
CMBReader.EVENT.CommandCompleted
```

CMBReader.EVENT.ReadResultReceived
This event is triggered whenever a scan result is received. A result is a CMBReadResults object (see CMBReadResults class

reference).

CMBReader.EVENT.AvailabilityChanged
This event is triggered when the availability of the ReaderDevice changes (example: when the MX Mobile Terminal has connected or disconnected the cable, or has turned on or off). The result is an int containing the availability information.

CMBReader.EVENT.ConnectionStateChanged
This event is triggered when the connection state of the ReaderDevice changes. The result is an int containing the connection information.

CMBReader.EVENT.ScanningStateChanged
This event is triggered when the scanner state of the ReaderDevice changes. The result is a boolean that is true if the scanning started, or false if it stopped.

CMBReader.EVENT.CommandCompleted
This event is triggered when a ReaderDevice command has completed. The result contains the following information:

```
commandID (String, the same param that was used to send the command)
eventType (String, ex: isSymbologyEnabled)
command (String, the command that was sent)
success (Boolean)
status (nullable, int, command status (See CDMResponse.h))
message (nullable, String, command payload)
image (nullable, base64 String representation of the scan image)
response (nullable, Boolean, command response, ex: isSymbologyEnabled will return true/false here)
```