

## Nativescript (v2.7.x)

### Changelog

Detailed changelog can be found [here](#).

### Install cmbSDK Nativescript plugin in your application

From the command prompt go to your app's root folder.

You can use our plugin from npm (recommended) or you can download from [here](#) and use from local path.

```
tns plugin add cmbsdk-nativescript # or local path
```

### Implement cmbSDK Nativescript plugin in your application

The best way to explore the usage of the plugin is to check our demo app. You can download our demo app from [here](#).

Once you download navigate to demo app root folder and run demo application:

```
cd cmbSDK_Nativescript/demo  
tns run android # or ios
```

In short to use our plugin in your project here are the steps:

#### Import the plugin

- [Typescript](#)

```
import { CMBReader, CMBReaderConstants } from 'cmbsdk-nativescript';
```

#### Set necessary callbacks and configure reader device

Open **home-view-model.ts** from demo app to check this code. All code in our demo app is with short description.

#### Start scanning process

```
this.cmb.startScanning()
```

## License Key(s)

### IMPORTANT

Usage of the cmbSDK nativescript plugin with an MX device is free, but if you want to utilize the CAMERA DEVICE (scan with the smartphone camera), you need to obtain a license from [CMBDN](#).

The Reader still works without a license, but results are randomly masked with \* chars.

It's free to register and you can obtain a 30 day trial license key.

Once the key is obtained there are two ways to use it in your application.

- First way is to include your key from code using **registerSDK** API method. You need to call this method **before** loadScanner.

```
this.cmbReader.registerSDK("SKD_KEY");
```

- Second way is to include your key in AndroidManifest.xml file for Android

```

1  <?xml version='1.0' encoding='utf-8'?>
2  <manifest android:hardwareAccelerated="true" android:versionCode="10000" android:versionName="1.0.0" package="io.cordova.hellocordova"
3  <supports-screens android:anyDensity="true" android:largeScreens="true" android:normalScreens="true" android:resizeable="true"
4  <uses-permission android:name="android.permission.INTERNET" />
5  <application android:hardwareAccelerated="true" android:icon="@mipmap/icon" android:label="HelloCordova" android:supportsRtl="
6  <activity android:configChanges="orientation|keyboardHidden|keyboard|screenSize|locale" android:label="@string/launcher_name"
7  <intent-filter android:label="@string/app_name">
8  <action android:name="android.intent.action.MAIN" />
9  <category android:name="android.intent.category.LAUNCHER" />
10 </intent-filter>
11 </activity>
12 <activity android:label="HelloCordova" android:name="com.cognex.cmb.sampleapp.ScannerActivity" android:theme="@style/AppTheme"
13 <meta-data android:name="MX_MOBILE_LICENSE" android:value="s756DCbRKHioo/Om8ru4L80W64IS9YGI4LKw8PbuG/M=" />
14 </application>
15 <uses-sdk android:minSdkVersion="16" android:targetSdkVersion="26" />
16 <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
17 <uses-permission android:name="android.permission.CAMERA" />
18 <uses-feature android:name="android.hardware.camera" />
19 <uses-feature android:name="android.hardware.usb.host" />
20 </manifest>
21

```

and Info.plist file for iOS

The screenshot shows the 'HelloCordova-Info.plist' file in Xcode. The table below represents the content of this file.

| Key                                    | Type       | Value                                     |
|--|------------|---|
| Information Property List              | Dictionary | (22 items)                                |
| Localization native development re...  | String     | English                                   |
| Bundle display name                    | String     | HelloCordova                              |
| Executable file                        | String     | \$(EXECUTABLE_NAME)                       |
| Icon files (iOS 5)                     | Dictionary | (0 items)                                 |
| CFBundleIcons~ipad                     | Dictionary | (0 items)                                 |
| Bundle identifier                      | String     | io.cordova.hellocordova                   |
| InfoDictionary version                 | String     | 6.0                                       |
| Bundle name                            | String     | \$(PRODUCT_NAME)                          |
| Bundle OS Type code                    | String     | APPL                                      |
| Bundle versions string, short          | String     | 1.0.0                                     |
| Bundle creator OS Type code            | String     | ????                                      |
| Bundle version                         | String     | 1.0.0                                     |
| Application requires iPhone enviro...  | Boolean    | YES                                       |
| Main nib file base name                | String     |   |
| Main nib file base name (iPad)         | String     |   |
| Supported interface orientations       | Array      | (3 items)                                 |
| Supported interface orientations (i... | Array      | (4 items)                                 |
| UIRequiresFullScreen                   | Boolean    | YES                                       |
| App Transport Security Settings        | Dictionary | (1 item)                                  |
| Supported external accessory prot...   | Array      | (1 item)                                  |
| Privacy - Camera Usage Description     | String     | Required for Scanning                     |
| MX_MOBILE_LICENSE                      | String     | FK0w10SCSPewE0LerbpwCA/477TapMHkKoaSNUgQ= |

## API

### loadScanner

To get a scanner up and running, the first thing to do, is to call the **loadScanner()** method. It expects a **CMBReaderConstants.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

```

this.cmb.loadScanner(CMBReaderConstants.DEVICE_TYPE.MXReader)
  .then(result => {
    this.cmb.connect()
      .then(result => {
        ...
      })
    .catch(err => {
      console.log(err);
    });
  });

```

### 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:

```
this.cmb.connect()
  .then(result => {
    ...
  })
  .catch(err => {
    console.log(err);
  });
```

There is an Event Listener for the connection status of the ReaderDevice, namely the **CMBReaderConstants.EVENTS.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()** :

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

Similarly to **connect()**, **disconnect()** also triggers the **CMBReaderConstants.EVENTS.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 **CMBReaderConstants.EVENTS.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(err => {
  ...
});
```

**scanImageFromBase64** Declaration

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

## setSymbologyEnabled

Once there is a connection to the Reader, we can enable symbologies by calling **setSymbologyEnabled()**. It expects three params: a `CMBReaderConstants.SYMBOLLOGY` which is the symbology to be enabled or disabled and a boolean for ON/OFF.

```
this.cmb.setSymbologyEnabled(CMBReaderConstants.SYMBOLLOGY.DataMatrix, true)  
  .then(function (result) {  
    if (result == true)  
      console.log("DataMatrix enabled");  
    else  
      console.log("DataMatrix NOT enabled");  
  })  
  .catch(function (err) {  
    console.log(err);  
  });
```

## isSymbologyEnabled

To check if we have a symbol enabled, we use **isSymbologyEnabled()**. It takes argument `CMBReaderConstants.SYMBOLLOGY`.

```
cmb.isSymbologyEnabled(CMBReaderConstants.SYMBOLLOGY.QR).then(function (result) {  
  if (result == true)  
    console.log("DataMatrix enabled");  
  else  
    console.log("DataMatrix NOT enabled");  
})  
.catch(function (err) {  
  console.log(err);  
});
```

## setCameraMode

This should be used only when using the device's built in camera for scanning (`CMBReaderConstants.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 `CMBReaderConstants.CAMERA_MODE` argument.

```
cmb.setCameraMode(CMBReaderConstants.CAMERA_MODE)  
/**  
Use camera with no aimer. Preview is on, illumination is available.  
CMBReaderConstants.CAMERA_MODE.NoAimer = 0,  
Use camera with a basic aimer (e.g., StingRay). Preview is off, illumination is not available.  
CMBReaderConstants.CAMERA_MODE.PassiveAimer = 1,
```

```
Use mobile device front camera. Preview is on, illumination is not available.  
CMBReaderConstants.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(mask, flag)  
.then(result => {  
  ...  
})  
.catch(err => {  
  ...  
});
```

```
cmb.disableCameraFlag(mask, flag)  
.then(result => {  
  ...  
})  
.catch(err => {  
  ...  
});
```

## setPreviewOptions

This should be used only when using the device's built in camera for scanning (CMBReaderConstants.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.

```
this.cmb.setPreviewOptions(CMBReaderConstants.CAMERA_PREVIEW_OPTION.NoZoomBtn | CMBReaderConstants.CAMERA_PREVIEW_OPTION.NoIllumination);
```

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 (CMBReaderConstants.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.

```
this.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 (CMBReaderConstants.DEVICE\_TYPE.Camera).

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

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

## setPreviewContainerBelowStatusBar

This should be used only when using the device's built in camera for scanning (CMBReaderConstants.DEVICE\_TYPE.Camera).

Available only on iOS.

Sets the camera preview partial view top axis to start below the status bar matching the Androids behaviour. 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(CMBReaderConstants.OVERLAY_MODE.OM_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.

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

## enableImageGraphics

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

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



## setParser

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

```
this.cmb.setParser(CMBReaderConstants.RESULT_PARSER.GS1);
```

## setReadStringEncoding

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

```
this.cmb.setReadStringEncoding(CMBReaderConstants.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](#).

```
this.cmb.sendCommand("GET_DEVICE.TYPE")
  .then(function (result) {
    console.log(result);
  })
  .catch(function (result) {
    console.log(result);
  });
```

## 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](#)

```
this.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](#)

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

## resetConfig

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

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

```
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.

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

## getSdkVersion

To see what cmbSDK version you're currently using:

```
this.cmbSdkVersionText = this.cmb.getSdkVersion();
```

## getCameraExposureCompensationRange

```
/* @return
   (promise) {
     range: JSON object with these attributes
           "lower" : min camera exposure value
           "upper" : max camera exposure value
           "step"  : camera exposure step value
     err : string , in case of error (e.g. reader not initialized)
   }
 */
```

```
cmb.getCameraExposureCompensationRange().then(function (range) {
  console.log(range);
})
.catch(function (err) {
  console.log(err);
});
```

**Note:** The camera needs to be started within cmbSDK at least once to get the camera exposure compensation range, otherwise it will return empty json.

## setCameraExposureCompensation

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

```
cmb.setCameraExposureCompensation(5)
.catch(function (err) {
  console.log(err);
});
```

**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(mask, param, value)
.then(result => {
  ...
})
.catch(err => {
  ...
});
```

## 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()
.catch(err => {
  ...
});
```

**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 == CMBReaderConstants.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 CMBReaderConstants.CONNECTION_STATE value of the current reader device  
*/
```

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

```
this.cmb.getConnectionState().then(function(connectionState){  
  if (connectionState == CMBReaderConstants.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 cmbSDK plugin emits Events that can be used in the application.

First, create the event emitter:

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

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

```
this.cmb.on(CMBReaderConstants.EVENTS.AvailabilityChanged, (args: any) => {
  if (args.data == CMBReaderConstants.AVAILABILITY.Available) {
    this.cmb.connect();
  } else {
    dialogs.alert({
      message: "Device became unavailable",
      okButtonText: "OK"
    });
  }
});
```

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

```
CMBReaderConstants.EVENT.ReadResultReceived
CMBReaderConstants.EVENT.AvailabilityChanged
CMBReaderConstants.EVENT.ConnectionStateChanged
CMBReaderConstants.EVENT.ScanningStateChanged
```

### **CMBReaderConstants.EVENT.ReadResultReceived**

This event is triggered whenever a scan result is received. A result is a [CMBReadResults class reference](#).

### **CMBReaderConstants.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.

### **CMBReaderConstants.EVENT.ConnectionStateChanged**

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

### **CMBReaderConstants.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.