Apache Cordova™ Plugin (v2.6.x)

Introduction

The purpose of these wiki pages is to provide detailed description of the API methods of the **Cordova** plugin that implements the cmbSDK.

Platform Targets

The plugin on the JavaScript side of things, is implemented as one *js* file that can be found in the *www* folder of your **Cordova** app.

> Supports the two major platforms: **Android** and **iOS.** All features on both platforms are tailored to work the same from a user/developer point of view.
> Supports two modes of scanning: With a smartphone camera, which utilizes the Manatee Works Barcode Scanner SDK (property of Cognex) and with a dedicated hardware scanner from the MX* series (hardware designed and produced by Cognex).

Impementation

In the plugin folder, we also provide a sample folder, which is basically three files that can be copied to your **js/index.js css/index.css** and **index.html** of your project and a **cmbconfig.js** file that is copied automatically to the **/js** folder, when the plugin is installed.

It covers other implementations too, specifically **Ionic** and a Hybrid app approach based on **SAP/FIORI** platform.

Installation

To build an awesome app with **Cordova**, we first need to create one with the cordova starter:

```
cordova create awesome-cordova-app-with-cmbsdk
cd awesome-cordova-app-with-cmbsdk
cordova plugin add absolute_path_to_plugin_directory          //for example: /Users/superUser/cmb-cordova-master
cordova platform add android
//cordova platform add ios //if you want to build on ios
```

For an **Ionic** solution:

```
#install ionic
sudo npm install -g cordova ionic
#start an ionic app with tabs layout
ionic start awesome-ionic-app tabs
cd awesome-ionic-app
#add our plugin
ionic cordova plugin add /WORKPLACE/PLUGINS/cmb-cordova //path to where you unzipped our plugin
#add the platform
ionic cordova platform add android@7.0.0
brew update && brew install gradle
# see https://gradle.org/install/
//to build directly from the console need to change the permissions of the gradlew file
sudo chmod 755 /Users/**/platforms/android/gradlew
//going to need ionic-native too
npm install ionic-native --save

//will run on android with live refresh and console logs
ionic cordova run android -l -c
```

This will create an app. The developer needs to be familiar with the process of developing on **Cordova**. There are quirks, like **iOS** will want a signing profile, or **Android** will complain about the manifest file.

Once there is a **Cordova** app that builds on the desired platform, we can add our Cognex solution.

In the **Cordova** plugin folder from the sample folder we need to copy the *index.html* into the platform **www** folder. The **index.js** into **www/js/index.js***,* and the **index.css** into **www/css/index.css**. **cmbconfig.js** is copied automatically to the **www/js** folder.

Licensing the SDK

If you plan to use the cmbSDK to do mobile scanning with a smartphone or tablet (with no MX mobile terminal), then the SDK requires the installation of a license key. Without a license key, the SDK will still operate, although scanned results will be obfuscated (the SDK will randomly replace characters in the scan result with an asterisk character).

Contact your Cognex Sales Representative for information on how to obtain a license key including trial licenses which can be used for 30 days to evaluate the SDK.

Android - After obtaining your license key, add the following line in your application's AndroidManifest.xml file, under the application tag:

```
<meta-data android:name="MX_MOBILE_LICENSE" android:value="YOUR_MX_MOBILE_LICENSE"/>
```

Next, put your key in place of YOUR_MX_MOBILE_LICENSE.

```
<application android:hardwareAccelerated="true" android:icon="@mipmap/icon" android:label="@string/app_name" android:suppor
        <activity android:configChanges="orientation|keyboardHidden|keyboard|screenSize|locale" android:label="@string/acti
            <intent-filter android:label="@string/launcher_name">
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <meta-data android:name="MX_MOBILE_LICENSE" android:value="g/9ytJzcja+sxt4DTEDxR4hp6sZh9bmL97vUx+EE9uY="/>
    </application>
```

iOS - After obtaining your license key, add it as a String in your application's Info.plist file, under the key *MX_MOBILE_LICENSE*.

Changelog

Detailed changelog can be found here.

Constants

The list of Constants that can be used by a developer is as follows:

Devices

There are two types of devices that you can use with this **Cordova** plugin. An MX device which is a hardware scanner, and a smartphone Camera.

```
DEVICES : ["DEVICE_TYPE_MX_1000","DEVICE_TYPE_MOBILE_DEVICE"],
DEVICES_FRIENDLY : ["MX Device","Camera"],
```

Triggers

There are two types of triggers, but six values are reserved. For the **MANUAL_TRIGGER** setting, once a barcode is found during the scanning process, the scanner will automatically stop further scanning. For **CONTINUOUS_TRIGGER** it will keep looking for a barcode and will fire the event each time it finds a barcode.

```
TRIGGER_TYPES : ["","","MANUAL_TRIGGER","","","CONTINUOUS_TRIGGER"],
```

Availability

```
AVAILABILITY_UNKNOWN :  0,
AVAILABILITY_AVAILABLE :  1,
AVAILABILITY_UNAVAILABLE :  2,
```

Connection State

When we monitor the connection state of the READER device (either Camera or MX) with setConnectionStateDidChangeOfReaderCallback we can get one of the following integer values that can be accessed via the **CONSTANTS** object

```
CONNECTION_STATE_DISCONNECTED :  0,
CONNECTION_STATE_CONNECTING :  1,
CONNECTION_STATE_CONNECTED :  2,
CONNECTION_STATE_DISCONNECTING :  3,
```

Camera Modes

The developer has access to the following camera modes:

```
CAMERA_MODES : ["NO_AIMER", "PASSIVE_AIMER", "ACTIVE_AIMER", "FRONT_CAMERA"]
```

NO_AIMER : Use camera with no aimer. Preview is on, illumination is available.
PASSIVE_AIMER : Use camera with a basic aimer (e.g., StingRay). Preview is off, illumination is not available.
ACTIVE_AIMER : Use camera with an active aimer (e.g., MX-100). Preview is off, illumination is available.
FRONT_CAMERA : Use mobile device front camera. Preview is on, illumination is not available.

Preview Overlay Modes

The developer has access to the following preview overlay modes:

```
PREVIEW_OVERLAY_MODE : ["OM_CMB", "OM_LEGACY"],
```

OM_CMB: Use CMB overlay'd preview. Provides a scanner timeout indicator but doesn't provide a visual indicator of the decoder ROI (viewfinder area).
OM_LEGACY: Use MW (legacy) overlay'd preview. Provides a visual indicator of the decoder ROI (viewfinder area). Doesn't provide scanner timeout indicator.

If you are going to change the previewOverlayMode, you need to do it before loadScanner is called, otherwise it will not work properly.

Symbols

A list of available **SYMBOLS** to use with the our scanner.

Not all symbols are implemented on both Camera Scanner and MX* Scanner

Refer to the wiki pages <u>Appendix A DMCC for the Camera Reader</u> for more info on support for **SYMBOLS**.

```
SYMBOLS : {
        "SYMBOL.UNKNOWN"            : 0
       ,"SYMBOL.DATAMATRIX"         : 1
       ,"SYMBOL.QR"                 : 2
       ,"SYMBOL.C128"               : 3
       ,"SYMBOL.UPC-EAN"            : 4
       ,"SYMBOL.C11"                : 5
       ,"SYMBOL.C39"                : 6
       ,"SYMBOL.C93"                : 7
       ,"SYMBOL.I2O5"               : 8
       ,"SYMBOL.CODABAR"            : 9
       ,"SYMBOL.EAN-UCC"            : 10
       ,"SYMBOL.PHARMACODE"         : 11
       ,"SYMBOL.MAXICODE"           : 12
       ,"SYMBOL.PDF417"             : 13
       ,"SYMBOL.MICROPDF417"        : 14
       ,"SYMBOL.DATABAR"            : 15
       ,"SYMBOL.POSTNET"            : 16
       ,"SYMBOL.PLANET"             : 17
       ,"SYMBOL.4STATE-JAP"         : 18
       ,"SYMBOL.4STATE-AUS"         : 19
       ,"SYMBOL.4STATE-UPU"         : 20
       ,"SYMBOL.4STATE-IMB"         : 21
       ,"SYMBOL.VERICODE"           : 22
       ,"SYMBOL.RPC"                : 23
       ,"SYMBOL.MSI"                : 24
       ,"SYMBOL.AZTECCODE"          : 25
       ,"SYMBOL.DOTCODE"            : 26
       ,"SYMBOL.C25"                : 27
       ,"SYMBOL.C39-CONVERT-TO-C32" : 28
       ,"SYMBOL.OCR"                : 29
       ,"SYMBOL.4STATE-RMC"         : 30
}
```

Bar code Result Object

```
interface BarcodeResult {
 readString : string,
 symbologyString: string,
 symbology : integer,
 goodRead : bool,
 xml: string,
 imageGraphics? : string,
 image? : string
};
```

The result from a successful scan is returned in an JavaScript Object with the layout shown above. The image is optional and it depends on the <u>enableImage</u>() API method.

Api Methods

loadScanner()

cmbScanner.loadScanner(deviceType,[callback])

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

It expects a **device type** and a **callback** function as a second param. The **callback** function is wrapped within a Promise and it's returned as one.

> This method does not connect to the Reader Device. We need to call **connect()** in the **callback** (promise) to actually connect to the Reader Device.

```
/*   @return
{
    const (string)action : the taken action (will always return LOAD READER)
    (string)  result : the message from the server
    (bool)    status : if the reader was loaded it will return true
    (string)  err    : the string error if an error was thrown
    (int)     type   : the type of the device that we connected to [0,1]
    (string)  name   : the name of the type of device DEVICES[type]
}
*/
```

The device type parameter can either be a string constant, with possible values:

```
DEVICES : ["DEVICE_TYPE_MX_1000","DEVICE_TYPE_MOBILE_DEVICE"]
```

Or we can just pass 0 for DEVICE_TYPE_MX and 1 for DEVICE_TYPE_MOBILE_DEVICE

Examples

```
//example as a callback:
    cmbScanner.loadScanner("DEVICE_TYPE_MOBILE_DEVICE",function(result){
        cmbScanner.connect();
    });
//example as a Promise
    cmbScanner.loadScanner("DEVICE_TYPE_MOBILE_DEVICE").then(function(result){
        cmbScanner.connect();
    });
//example passing the device type as an integer
    cmbScanner.loadScanner(0).then(function(result){
        cmbScanner.connect();
    });
```

connect()

To connect to a READER after we've loaded it, we need to use **Connect**.

```
cmbScanner.connect([callback]);
```

```
/*   @return
    (promise)  {
        status : boolean, if connection succeded 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 return the result of the connection attempt.

Examples

```
/** Using connect and returning the value as a promise */
cmbScanner.connect().then(function(connectionState){
    console.log(connectionState);
// { "status" : false, "err" : "Did not return OK in success" }
});
/** Using connect and returning the value as a callback function */
cmbScanner.connect(function(connectionState){
  console.log(connectionState);
});
```

Even though we return the result of the **connect** action, acting on the result from the connect callback for the purpose of updating DOM elements or calling API methods that only work after we have a connection set, is not advised.

There is a listener callback **(**setConnectionStateDidChangeOfReaderCallback**)** function that can be set, that will always trigger whenever there is a change in the connection state and we should really keep that code there.

disconnect()

```
/*   @return
    (promise)  {
        status : boolean, if connection succeded 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 opossite of **connect()** :

```
cmbScanner.disconnect([callback])
```

Similarly to **connect()**, **disconnect()** too returns the connectionState in the callback function (which is wrapped into a Promise), and we could act on the connectionState, for example change the label from "connected" to "disconnected":

```
cmbScanner.disconnect(function(connectionState){
  if(connectionState == cmbScanner.CONSTANTS.CONNECTION_STATE_DISCONNECTED){

    document.getElementById('some-label').innerHtml('DISCONNECTED');
  }
  else if (connectionState == cmbScanner.CONSTANTS.CONNECTION_STATE_CONNECTED){
    document.getElementById('some-label').innerHtml('DISCONNECTED');
```

```
    }
    })
```

But, just like **connect()**, we don't have to do this because of **setConnectionStateDidChangeOfReaderCallback()** that sets a listener to all connect / disconnect / connecting / disconnecting events.

setResultCallback()

To handle successful scan results, we need to setup the **ResultCallback** function. This is done via the **setResultCallback**.

> **This method DOESN'T return a PROMISE.**

It sets the callback function for all **didReceiveReadResultFromReader** events.

> **cmbScanner.setResultCallback(callback)**

```
cmbScanner.setResultCallback(function(result){
     if(result && result.readResults && result.readResults.length > 0){

               result.readResults.forEach(function (item, index){

                    if (item.goodRead == true) {
//Perform some action on barcode read
//example:
                         document.getElementById('content').insertAdjacentHTML('beforeend','<div class="result"><span class
//we could put all this DOM handling in the dom helper object, but since it's just one line of code let's leave it be
                    }
                    else{
//Perform some action when no barcode is read or just leave it empty
          // navigator.notification.alert("Stopped");
                    }
               });
          }
});
```

ResultCallback will fire every time there is a barcode scan (or the scanner was stopped either manually or it timedout with No Result).

Result Object

The structure of the barcode result is as follows:

> - **result.readResults** - json array. If you use multicode mode here you will find main result(set of all partial results together merged in one readString) and all other partial results
> - **result.subReadResults** - json array of all partial results (if single code mode is uset this array will be empty)
> - **result.xml** - string representation of complete result from reader device in xml format
>
> **result.readResults** and **result.subReadResults** are json arrays that contains items with this structure:
> - **item.readString** - string representation of barcode
> - **item.symbologyString** - string representation of the barcode symbology detected
> - **item.goodRead** - bool that indicate if barcode is successful scanned
> - **item.xml** - string representation of partial result in xml format
> - **item.imageGraphics** - string that represent svg image from last detected frame
> - **item.image** - base64 string that contain image from last detected frame

setAvailabilityCallback()

To monitor for availability of the MX device use:

setAvailabilityCallback([callback])

Set a listener on availability changes. Will fire when the device becomes available or loses availability.

Does not return a Promise.

Usage

```
cmbScanner.setAvialabilityCallback(funciton(availability){
        /**
        * availability - int representation of the Device availability
        */
        if (availability == CONSTANTS.AVAILABILITY_UNKNOWN) {
            //Perform some action when device availability is not known
        }
        else if (availability == CONSTANTS.AVAILABILITY_AVAILABLE) {
            //Perform some action when device is available
        }
        else if (availability == CONSTANTS.AVAILABILITY_UNAVAILABLE) {
            //Perform some action when device is not available
        }
});
```

setConnectionStateDidChangeOfReaderCallback()

cmbScanner.setConnectionStateDidChangeOfReaderCallback([callback])

All **connect/disconnect** events should be handled within the callback function set with this API method.

This method DOESN'T return a PROMISE. It sets the callback function for all *connectionStateDidChangeOfReader* events.

The callback is optional, because the plugin provides a default callback that handles **connect/disconnect** events, but the default callback is only a placeholder function and doesn't offer much functionality to the end app developer.

The App developer should **always set** this method, as most of the configuration settings for the Reader will be done within this listener.

When the **connetionStateDidChange** is fired, it will return one of 4 possible integer values

CONNECTION_STATE_DISCONNECTED : 0,
CONNECTION_STATE_CONNECTING : 1,
CONNECTION_STATE_CONNECTED : 2,
CONNECTION_STATE_DISCONNECTING : 3

These can be accessed directly as integers, or the developer can use them from the CONSTANTS enum available to them by accessing the CONSTANTS object within the cmbScanner**.**

cmbScanner.CONSTANTS.CONNECTION_STATE_CONNECTED

Example of this in action would be:

```
cmbScanner.setConnectionStateDidChangeOfReaderCallback(function(connectionState){
      if(connectionState == cmbScanner.CONSTANTS.CONNECTION_STATE_CONNECTED){
//do stuff while connected, like set a symbology to enabled
          return cmbScanner.setSymbologyEnabled("SYMBOL.QR",true).then(function(rr){
//see setSymbologyEnabled for more info on the rr (returned result) object
```

```
    //after the symbol has been set we can send a command. Let's set the flash ON
                cmbScanner.sendCommand("SET LIGHT.INTERNAL-ENABLE ON")
                 .then(function(result){
    //if the command is succesful we should have the LIGHT turn ON whenever we start the scanning process
    //and to check if the flash is ON we can use the isLightsOn API method
                     cmbScanner.isLightsOn().then(function(lights_on){
    //light should be on
                     });
                });
            });
        }
    });
```

Here we not only listen to the **connect/disconnect** event, but we also enable a few symbologies and set the flash light on, via a command and the API method **isLightsOn**.

setActiveStartScanningCallback()

If we want to monitor the state of the scanner (READER), we need to setup this callback function.

The callback function will be fired every time the scanner starts looking for a bar code (either via a Camera Reader, or via the trigger button on an MX device) or stops looking (via a manual stopScanning() call or if it finds a bar code).

Declaration

cmbScanner.setActiveStartScanningCallback(callback)

Usage

```
cmbScanner.setActiveStartScanningCallback(function(scannerState){
  if(scannerState == true){
    console.log('scanner is working');
  }
  else{
    console.log('scanner is not working');
  }
  });
```

As previously mentioned, setting this function, completely removes the need to setup a callback on the **startScanning()** or **stopScanning()** methods.

Even though it may seem more appropriate to just have that action return in a promise or a callback immediately after **startScanning()/stopScanning()** is called, due to the nature of the **stopScanning** process, which occurs every time there's is a result, it's best to listen for both actions (starting and stopping) in the same place.

checkCameraPermission()/requestCameraPermission()

To check and request permission to use the camera for scanning barcodes with the mobile device.

checkCameraPermission Declaration

void *cmb.checkCameraPermission(imageUri, callback)*

requestCameraPermission Declaration

void *cmb.requestCameraPermission(base64, callback)*

```
/*   @return
    (promise)  {
        status : boolean, status of permission to use camera
        message : int, 1 if permission can be requested, 0 otherwise
    }
*/
```

startScanning()/stopScanning()

To start/stop the scanning process, we use these two methods. They can either be called from a button via the app's UI, or we can call the programmatically.

startScanning Declaration

void *cmbScanner.startScanning([callback])*

stopScanning Declaration

void *cmbScanner.stopScanning(callback)*

Usage

They take an optional callback as a parameter, but it will overwrite the "main" callback function only if it's not set by *setActiveStartScanningCallback()* already.

Because of the nature of the READERS we should always set the **setActiveStartScanningCallback()** function, if we want to control certain DOM elements based on the status of the scanner (Reader).

```
cmbScanner.startScanning(function(scannerState){
 if(scannerState)
  document.getElementById('scanner-active-label')[removed] = 'RUNNING';
 else
  document.getElementById('scanner-active-label')[removed] = 'STOPPED';
});
```

Examples

```
<a href="cmbScanner.startScanning();">Start Scanner</a> <!-- button to start the scanner -->
<a href="cmbScanner.stopScanning();">Stop Scanner</a> <!-- button to stop the scanner -->

<a href="cmbScanner.startScanning(function(scanner_state){
//this callback function will be used, only if we don't already have one set via
// setActiveStartScanningCallback. In all other cases it will be ignored.
console.log(scanner_state);
});">Start Scanner</a> <!-- button to start the scanner with a callback, avoid this and always use setActiveStartScanningCa
```

scanImage()/scanImageFromBase64()

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

scanImage Declaration

void *cmb.scanImage(imageUri, callback)*

scanImageFromBase64 Declaration

void *cmb.scanImageFromBase64(base64, callback)*

```
/*   @return
    (promise)  {
        status : boolean, status of image scanning execution
        err : string , if status false err will not be null
    }
*/
```

setSymbologyEnabled()

Once there is a connection to the Reader, we can enable Symbols by calling

setSymbologyEnabled(symbol,on_off,[callback]).

If Mobile Camera is used as a Reader, there are **NO SYMBOLS enabled by default.** We need to enable each SYMBOL that we want to use**.**

It expects a string value of the symbol to be enabled, see Constants for more information on how to use SYMBOLS.

Returns a Promise.

```
setSymbologyEnabled(symbol,on_off,[callback])

/* @return A promise that contains the JSON object
        {
            action  : the DMCC command that was invoked
            status  : did it succeed or not, if an error happened it will be set to false
            result   : if the symbol is enabled then true, if not then false
            err     : the error message if the action didn't complete
        }
*/
```

Example of how you would use it:

```
cmbScanner.setSymbologyEnabled("SYMBOL.C93",true,function(result){
  console.log(JSON.stringify(result));
//will print {"action" : "SET SYMBOL.C93 ON","status" : true, "result" : true, "err" : null}
});

//or with a Promise

cmbScanner.setSymbologyEnabled("SYMBOL.C93",true).then(function(result){
  console.log(JSON.stringify(result));
    //will print {"action" : "SET SYMBOL.C93 ON","status" : true, "result" : true, "err" : null}
});
```

isSymbologyEnabled()

To check if we have a symbol enabled, we use **isSymbologyEnabled()**.

isSymbologyEnabled(symbol,[callback])

The result is returned in the callback which is then wrapped into a Promise.

```
/**
@return A promise that contains the object
{
    action  : the DMCC command that was invoked
    status  : did it succeed or not, if an error happened it will be set to false
    result  : if the symbol is enabled then true, if not then false
    err     : the error message if the action didn't complete
}
*/
```

Example

```
cmbScanner.isSymbologyEnabled("SYMBOL.C93",function(rr){
  console.log(JSON.stringify(rr));
//will print {"action" : "GET SYMBOL.C93","status" : true, "err" : null,"result" : true}
});
```

setCameraMode()

To set how the camera will behave when we use camera device as a barcode reader we use:

(void) *cmbScanner.setCameraMode(cameraMode)*

Note: It should be called before we call **loadScanner()** for it to take effect. Calling it after the scanner was loaded won't do anything if the scanner is loaded.

```
/**
Use camera with no aimer. Preview is on, illumination is available.
NO_AIMER = 0,
```

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

Use camera with an active aimer (e.g., MX-100). Preview is off, illumination is available.
ACTIVE_AIMER = 2,

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

setPreviewOptions()

This controls the preview/scanning options when using the Mobile device camera. Multiple options can be OR'd together.

(void) *cmbScanner.setPreviewOptions(previewOptions)*

Note: It should be called before we call **loadScanner()** *for it to take effect. Calling it after the scanner was loaded won't do anything if the scanner is loaded.*

```
/**
 * Use defaults (no overrides).
 */
"DEFAULTS" : 0,
/**
 * Disable zoom feature (removes zoom button from preview).
 */
"NO_ZOOM_BTN" : 1,
/**
 * Disable illumination (removes illumination button from preview).
 */
"NO_ILLUM_BTN" : 2,
/**
 * Enables the simulated hardware trigger (the volume down button).
 */
"HARDWARE_TRIGGER" : 4,
/**
 * When scanning starts, the preview is displayed but decoding is paused until a trigger (either the on screen button or th
 */
"PREVIEW_PAUSED" : 8,
/**
 * Force the preview to be displayed, even if off by default (e.g., when using kCDMCameraModePassiveAimer or kCDMCameraMode
 */
"PREVIEW_ALWAYS_SHOW" : 16,
/**
 * Use higher resolution if the device supports it. Default is 1280x720, with this param 1920x1080 will be used.
 */
"HIGH_RESOLUTION" : 64,
/**
 * Use higher framerate if the device supports it. Default is 30 FPS, with this param 60 FPS will be used.
 */
"HIGH_FRAME_RATE" : 128,
/**
 * Show close button in partial view.
 */
"SHOW_CLOSE_BUTTON" : 256,
/**
 * Keep the preview in paused state after read or timeout.
 */
"KEEP_PREVIEW_IN_PAUSED_STATE" : 512
```

setPreviewContainerPositionAndSize()

To set the size and position of the preview container (only available to the user when they use the camera reader):

> (void) *setPreviewContainerPositionAndSize(x,y,w,h)*

```
/*
  @params x,y,w,h
        x,y  : top left position
        w,h  : width and height of the rectangular in precentages of the full container
*/
```

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

setPreviewContainerFullScreen()

Sets the camera preview to start in full screen instead of partial view. (only available to the user when they use the camera reader):

> (void) *setPreviewContainerFullScreen()*

setPreviewContainerBelowStatusBar()

Available only on iOS, applies only for the camera reader.

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

```
cmb.setPreviewOverlayMode(cmb.CONSTANTS.PREVIEW_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.

void *cmb.showToast(message)*

void *cmb.hideToast()*

setLightsOn()

To enable the flash light programmatically, we can use:

setLightsOn(lights_on, callback)

```
/**
@return A promise that contains the JSON object
{
  (string)  action  : the DMCC command that was invoked
  (bool)    status  : did it succeed or not, if an error happened it will be set to false
  (string)  err     : the error message if the action didn't complete
  (bool)  result    : the result of the taken action, in this case TRUE if the light was enabled, false if not
}
*/
```

Example

```
cmbScanner.setLightsOn(true,function(response){
  //console.log(response);
});

cmbScanner.setLightsOn(true).then(function(response){
//or with a promise, the callback function is wrapped within a promise
//so if we don't set a callback function as a param, the default one will be used
//and it will return a promise to continue the chain
});
```

isLightsOn()

To check if the flash light is ON, we can use:

> isLightsOn(callback)

```
/**
@return A promise that contains the JSON object
{
  (string)  action  : the DMCC command that was invoked
  (bool)    status  : did it succeed or not, if an error happened it will be set to false
  (string)  err     : the error message if the action didn't complete
  (bool)  result    : the result of the taken action, in this case TRUE if the light was enabled, false if not
}
*/
```

Example

```
cmbScanner.isLightsOn(true,function(response){
  //console.log(response);
});

cmbScanner.isLightsOn(true).then(function(response){
//or with a promise, the callback function is wrapped within a promise
//so if we don't set a callback function as a param, the default one will be used
//and it will return a promise to continue the chain
});
```

enableImage()/enableImageGraphics()

If we want to return the last image seen by the decoder we use **enableImage()**. If we want image graphics stats on the returned image we use **enableImageGraphics().**

Declaration

> **cmbScanner.enableImage(on_off_switch)**; **cmbScanner.enableImageGraphics(on_off_switch)**;

Usage

```
cmbScanner.enableImage(true);
//it will enable the image in the result


cmbScanner.enableImageGraphics(true);
//will show where exactly the barcode was detected on the image
```

The structure of the bar code result object is:

```
interface BarcodeResult {
  readString : string,
  symbologyString: string,
```

```
  symbology : integer,
  goodRead : bool,
  xml: string,
  imageGraphics? : string,
  image? : string
};
```

If set, the returning image is returned as base64 png image.

setParser()

Enable or disable parsing for scanned barcodes. Expects one argument of type cmb.CONSTANTS.PARSERS.

```
cmb.setParser(cmb.CONSTANTS.PARSERS.GS1);
```

setReadStringEncoding

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

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

sendCommand()

Most of the API methods can be replaced with just sending the appropriate commands to the MX (or Camera) reader.

(Promise) **sendCommand([callback])**

Send command receives a callback and wraps it into a promise, so the chain can be continued.

Usage

```
cmbScanner.sendCommand("SET SYMBOL.POSTNET OFF")
          .then(function(result){
//and in the promise let's see what our command did
                console.log(JSON.stringify(result));
          });
//or like this
cmbScanner.sendCommand("SET LIGHT.INTERNAL-ENABLE ON")
          .then(function(result){
//if the command is succesful we should have the LIGHT turn ON whenever we start the scanning process
                console.log(JSON.stringify(result));
          });
```

The first command will disable the **SYMBOL POSTNET**. The second will set the Flash ON.

Both actions can be performed with their appropriate API functions **setSymbologyEnabled()** and **setLightsOn()** and generally speaking a developer should always prefer the API functions to the **sendCommand().**

However, there are cases where the Cordova plugin doesn't cover certain use case scenarios, and we can always use the **sendCommand** for that.

Example

If we want to change the way our MX device "beeps" and "lights" we can use:

ACTION	COMMAND	PARAMETER	PARAM VALUES	DESCRIPTION OF PARAMETERS	RESULTING ACTION
SET|GET	OUTPUT.USER-CONFIGURE	event number, beep tone, beep number, LED color, vibration	[1,2] [0-2] [0-3] [0-7] [ON|OFF]	1: USER_EVENT_1, 2: USER_EVENT_2 [0]: Low, [1]: Medium, [2]: High Number of beeps [0]: Off, [1]: Blue, [2]: Green, [3]: Cyan, [4]: Red, [5]: Magenta, [6]: Yellow, [7]: White enable/disable ON: The reader vibrates. OFF: The reader does not vibrate.	Sets the beeper tone and the number of beeps for USER_EVENT_1 or USER_EVENT_2.

> SET OUTPUT.USER-CONFIGURE

So for our example we would have:

```
cmbScanner.sendCommand("SET OUTPUT.USER-CONFIGURE 1 1 2 4 ON").then(function(response){
  //console.log(response); //print out the result of the action
});
```

This will configure user event #1 to two medium beeps with red LED and vibration. In order to activate this event, we use:

```
cmbScanner.sendCommand("OUTPUT.USER1").then(function(response){
     //this will call the stored settings item for USER1, so the MX device will basically beep twice with red LED and vibra
})
```

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()

If we need to reset the configuration settings stored for the MX/Camera reader we can use:

Declaration

resetConfig([callback])

Usage

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

Result Interface

```
/*
@return A promise that contains the JSON object
{
  (string) action  : the DMCC command that was invoked
  (bool)   status  : did it succeed or not, if an error happened it will be set to false
  (string) err     : the error message if the action didn't complete and there is an error. default is null
  (bool)  result   : the result of the sendCommand action
}
*/
```

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.

registerSDK("SDK_KEY")

Example

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

getSdkVersion()

To see what cmbSDK version you're currently using:

> **getSdkVersion(callback)**

```
/**
@return A promise that contains the cmbSDK version string
{
   (string)  version  : cmbSDK version
}
*/
```

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.

```
cmbScanner.getCameraExposureCompensationRange(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));
}, function(error){ console.log(error); });
```

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.

```
cmbScanner.setCameraExposureCompensation(5, function(error){
    console.log(error);
});
```

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.

Migration from mwSDK to cmbSDK

Difference between mwSDK and cmbSDK

The Manatee Works Barcode Scanner SDK has been fully integrated into the Cognex Mobile Barcode SDK (cmbSDK). Therefore, we are shifting our focus to the cmbSDK.

Although the cmbSDK is backward compatible with the MW SDK, the Cordova API methods are different. Your account, login, license(s), and key(s) remain the same. The cmbSDK adds a higher-level API to the scanning methods that utilize the camera of a smartphone or tablet, so you will have to adapt your application to utilize the new API methods. With that you will have the added benefit of your app(s) supporting the Cognex MX Series mobile barcode readers, and MX Series mobile terminals, with a single code base.

Remove mwSDK

To avoid collision between mwSDK and cbmSDK, first you need to remove the ManateeWorks Cordova plugin. You can do that with the following Cordova command:

```
cordova plugin remove manateeworks-barcodescanner-v3
```

Add cmbSDK

Next add the cmbSDK Cordova plugin to your project.

Please navigate to this url to check step by step how to integrate cmbSDK inside your Cordova project. You can also add the plugin with the following Cordova command:

```
cordova plugin add cmbsdk-cordova
```

After that please remove all API's and methods that you are using from mwSDK, and follow our guide from here to see how to implement cmbSDK in your project.

Here are some of the main differences in code between mwSDK and cmbSDK:

1. **Initialize, create and connect reader device**
   - Using mwSDK we only use API methods from the **BarcodeScanner** object to initialize decoder before starting the scanner process: set active codes, set scanning rect, set decoder level, register sdk, etc. When we use cmbSDK all this things are done in code behind with default values by using a ReaderDevice object. Using cmbSDK, only creating a reader device in not enough to start scanning process, we also need to connect to the reader device and set necessary callbacks that will handle response from connection state changed, availability, result received, etc.

2. **Start scanning process**
   - With mwSDK after we initialize decoder we are ready to start the scanning. There are two different methods for starting the scanner. One for partial view (where we can set size as input parameter) and one for full screen and both return a scan result in the callback. We can't start scanning process if we don't have valid connection to reader device. Using cmbSDK there is only one method to start the scanning process and comes from ReaderDevice object. In Cordova, that is:

   ```
   cmbScanner.startScanning(function(scannerState){
   ...
   });
   ```

   If we want to use full screen mode when we use the function setPreviewContainerFullScreen() after a successful connection, or if we want to configure the partial view, we use the function . Result from scanning process will be received in onReadResultReceived callback function setPreviewContainerPositionAndSize(x,y,w,h).

3. **Result received**
   - If we have successful read or we stop the scanning process and have no read, result object will be received in the result callback. To configure the result callback, the following function is used:

```
cmbScanner.setResultCallback(callback)
```

In cmbSDK result object is more extended than in mwSDK. From that object we can read our barcode result, symbology, image from last frame, SVG result, etc.