Flutter (v2.5.x)

License Key(s)

IMPORTANT

Usage of the cmbSDK flutter 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.

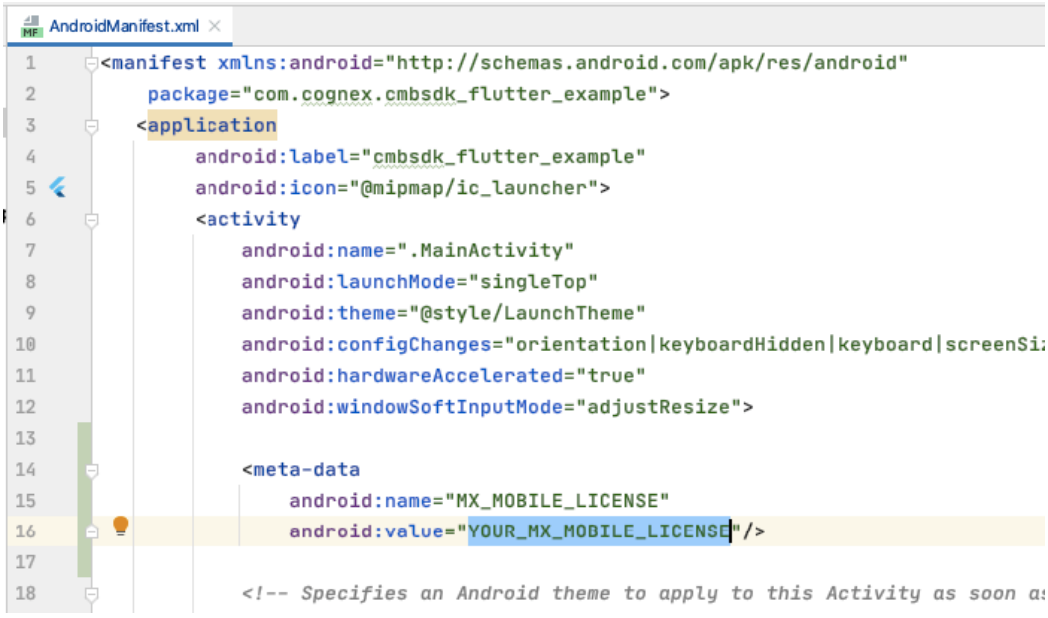
It's free to register and you can obtain a 30 day trial license key.

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

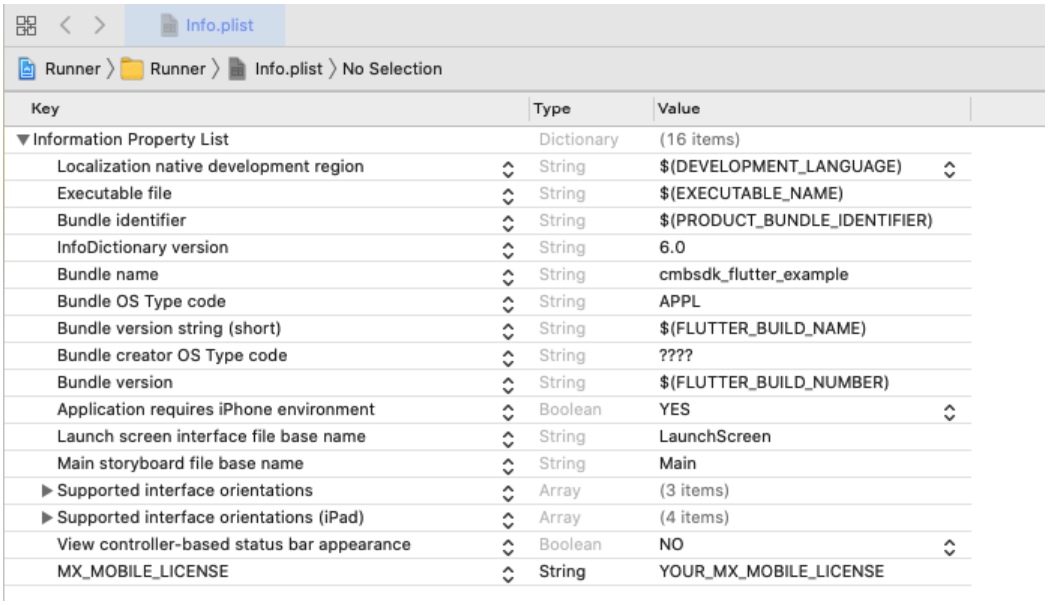
- First way is to include your key from code using **registerSDK** API method. You need to call this method **before** loadScanner.

```
CmbsdkFlutter.registerSDK("SKD_KEY");
```

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



and Info.plist file for iOS



Install cmbSDK flutter plugin in your application

You can find our plugin on pub.dev //todo set a link to our plugin, or you can use the local path to the plugin.

Open the **pubspec.yaml** file from your project and add a reference to the plugin.

If you use the plugin from the pub.dev //todo set a link to our plugin

```
dependencies:
  flutter:
    sdk: flutter

    cmbsdk_flutter: ^2.5.0
```

If you use from the local path

```
dependencies:
  flutter:
    sdk: flutter

    cmbsdk_flutter:
      path: {add path to the plugin}
```

Then you need to execute the **Pub get** command to pull the plugin into your project. From the command prompt go to your app's root folder and execute this command.

```
$ flutter pub get
```

If you use VS Code IDE, this command is automatically performed when you save the changes.

Implement cmbSDK flutter 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. //todo add a link to the flutter file

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

Import the plugin

```
import 'package:cmbsdk_flutter/cmbsdk_flutter.dart';
```

Set necessary callbacks and configure reader device

Open **main.dart** from the demo app to check this code. All code in our demo app is with a short description.

Start scanning process

```
CmbsdkFlutter.startScanning()
          .catchError((error, stackTrace) => print('${error.message}'));
```

API

loadScanner

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

```
CmbsdkFlutter.loadScanner(_deviceType).then((value) {
  CmbsdkFlutter.connect().then((value) {
     ...
  }).catchError((error, stackTrace) {
     ...
  });
});
```

connect

```
/*   @return
     - success callback if connection is completed
     - error callback if connection is rejected. The error object in this callback contains the error code and error messag

*/
```

The result from the connect() method is returned as a success or error callback

```
CmbsdkFlutter.connect().then((value) {
     ...
}).catchError((error, stackTrace) {
```

```
    ...
});
```

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

disconnect

```
/*   @return
     - success callback if disconnection is completed
     - error callback if disconnection is not successfull. The error object in this callback contains the error code and er

*/
```

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

```
CmbsdkFlutter.disconnect().then((value) {
    ...
}).catchError((error, stackTrace) {
    ...
});
```

Similarly to **connect()**, **disconnect()** also triggers the **ConnectionStateChangedListener** event.

startScanning / stopScanning

```
/*   @return
     - success callback if scanning state is changed
     - error callback if scanning state is not changed. The error object in this callback contains the error code and error
*/
```

To start / stop the scanning process, we use these methods. They return a success callback if the command was successful (the scanning has started or stopped) or error callback (if there is no active ReaderDevice initialized or isn't connected). These methods trigger the **ScanningStateChangedListener** event where current scanning state is returned (true if scanning process is started and false if scanning process is stopped).

```
CmbsdkFlutter.startScanning().then((value) {
    ...
}).catchError((error, stackTrace) {
    ...
});
```

After starting the scanner and scanning a barcode, the scan result triggers the **ReadResultReceivedListener** event.

setSymbologyEnabled

```
/*   @return
     - success callback if symbology is enabled
     - error callback if symbology is NOT enabled. The error object in this callback contains the error code and error mess
*/
```

Once there is a connection to the Reader, we can enable symbologies by calling **setSymbologyEnabled()**. It expects 2 params: a CmbsdkSymbology which is the symbology to be enabled or disabled and a boolean for ON/OFF.

```
CmbsdkFlutter.setSymbologyEnabled(CmbsdkSymbology.DataMatrix, true)
.then((value) => print('DataMatrix enabled'))
.catchError((error, stackTrace) =>
   print('DataMatrix NOT enabled. ${error.message}'));
```

isSymbologyEnabled

```
/*   @return
     - success callback with boolean value that show is symbology enabled or disabled
     - error callback if something went wrong. The error object in this callback contains the error code and error message
*/
```

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

```
CmbsdkFlutter.isSymbologyEnabled(CmbsdkSymbology.DataMatrix).then((value) {
      if (value)
        print('DataMatrix enabled');
      else
        print('DataMatrix NOT enabled');
}).catchError((error, stackTrace) =>
        print('${error.message}'));
```

setLightsOn

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

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

```
CmbsdkFlutter.setLightsOn(true)
   .catchError((error, stackTrace) => print('${error.message}'));
```

If we call this function before scanning is active, we are setting light initial state for every scanning session.

isLightsOn

```
/*   @return
     - success callback with boolean value that represent current light status
     - error callback if something went wrong. The error object in this callback contains the error code and error message
*/
```

We can check the lights status with isLightsOn(), which returns a callback with the current light status.

```
CmbsdkFlutter.isLightsOn().then((value) {
      if (value)
        print('Light is ON');
      else
        print('Light is OFF');
}).catchError((error, stackTrace) => print('${error.message}'));
```

setCameraMode

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

```
CmbsdkFlutter.setCameraMode(CmbsdkCameraMode.NoAimer);
```

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

setPreviewOptions

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

```
CmbsdkFlutter.setPreviewOptions(CmbsdkPrevewiOption.NoZoomBtn | CmbsdkPrevewiOption.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 (CmbsdkDeviceType.Camera).

setPreviewContainerPositionAndSize takes four integer arguments, 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.

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

setPreviewContainerFullScreen

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

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

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

```
CmbsdkFlutter.setPreviewContainerFullScreen()
.catchError((error, stackTrace) => print('${error.message}'));
```

setPreviewOverlayMode

Set the camera overlay mode.

Read more about overlay modes here and here

```
CmbsdkFlutter.setPreviewOverlayMode(CmbsdkOverlayMode.OMLegacy);
```

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

enableImage

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

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

```
CmbsdkFlutter.enableImage(true)
.catchError((error, stackTrace) => print('${error.message}'));
```

enableImageGraphics

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

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

```
CmbsdkFlutter.enableImageGraphics(true)
.catchError((error, stackTrace) => print('${error.message}'));
```

setParser

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

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

```
CmbsdkFlutter.setParser(CmbsdkResultParser.Gs1)
.catchError((error, stackTrace) => print('${error.message}'));
```

setReadStringEncoding

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

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

```
CmbsdkFlutter.setReadStringEncoding(CmbsdkReadStringEncoding.Utf8)
.catchError((error, stackTrace) => print('${error.message}'));
```

getDeviceBatteryLevel

```
/*   @return
     - success callback with value that represent the battery level
     - error callback if something went wrong. The error object in this callback contains the error code and error message
*/
```

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

```
CmbsdkFlutter.getDeviceBatteryLevel().then((value) {
      print('Battery level is $value');
}).catchError((error, stackTrace) => print('${error.message}'));
```

getAvailability

```
/*   @return
     - success callback with int value that represent the current availability status
```

```
     - error callback if something went wrong. The error object in this callback contains the error code and error message
*/
```

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

```
CmbsdkFlutter.getAvailability().then((availability) {
  if (availability == CmbsdkAvailability.Available.index)
    print('ReaderDevice is available');
  else
    print('ReaderDevice is NOT available');
}).catchError((error, stackTrace) => print('${error.message}'));
```

getConnectionState

```
/*   @return
     - success callback with int value that represent the current connection state status
     - error callback if something went wrong. The error object in this callback contains the error code and error message
*/
```

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

```
CmbsdkFlutter.getConnectionState().then((currentState) {
  if (currentState == CmbsdkConnectionState.Connected.index)
    print('ReaderDevice is connected');
}).catchError((error, stackTrace) => print('${error.message}'));
```

sendCommand

```
/*   @return
     - success callback that return result value from executed dmcc
     - error callback if something went wrong. The error object in this callback contains the error code and error message
*/
```

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. DMCC string is send as input parameter.

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

```
CmbsdkFlutter.sendCommand('GET DEVICE.TYPE')
.then((value) => print('$value'))
.catchError((error, stackTrace) => print('${error.message}'));
```

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

```
//TODO
```

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

```
//TODO
```

Listeners

AvailabilityChangedListener

This listener is used only for MXReader when the availability 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.

```
CmbsdkFlutter.setAvailabilityChangedListener((availability) {
  if (availability == CmbsdkAvailability.Available.index)
    print('ReaderDevice is available');
});
```

ConnectionStateChangedListener

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

```
CmbsdkFlutter.setConnectionStateChangedListener((state) {
  if (state == CmbsdkConnectionState.Connected.index)
```

```
    print('ReaderDevice is connected');
});
```

ScanningStateChangedListener

This listener 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.

```
CmbsdkFlutter.setScanningStateChangedListener((scanningState) {
   if (scanningState)
     print('Scanning is started');
});
```

ReadResultReceivedListener

This listener is triggered whenever a scan result is received.

```
CmbsdkFlutter.setReadResultReceivedListener((resultJSON) {
  final Map<String, dynamic> resultMap = jsonDecode(resultJSON);
});
```

A resultJSON is a JSON object with the following structure:

- **xml**: string representation of complete result from reader device in xml format

- **results**: json array of all results. If you use multicode mode here you will find main result and all other partial results.

- **subResults**: json array of all partial results (if single code mode is used this array will be empty)

results and subResults array contains items with this structure:

- **symbology**: int representation of the barcode symbology detected

- **symbologyString**: string representation of the barcode symbology detected

- **readString**: string representation of barcode

- **goodRead**: bool that indicate if barcode is successful scanned

- **isGS1**: bool that indicate if barcode is GS1 or not

- **parsedText**: string that represent parsed text from the result

- **parsedJSON**: string that represent parsed text in json format from the result

- **imageGraphics**: string that represent svg image from last detected frame

- **image**: base64 string that contain image from last detected frame

- **xml**: string representation of partial result in xml format