Cognex Mobile Barcode SDK for iOS (v2.4.x)

Overview

Cognex Mobile Barcode SDK (cmbSDK) is a tool for developing mobile barcode scanning applications. CmbSDK is based on Cognex's DataMan technology and the Manatee Works Barcode Scanning SDK allowing you to create applications for barcode scanning on mobile devices. Mobile devices used for barcode scanning are supported smartphones, tablets and the MX Series industrial barcode readers. CmbSDK abstracts the device through a *CMBReaderDevice* connection layer. Once the application establishes connection with the reader, a single and unified API serves as an interface to configure the device, without writing too much conditional code.

CmbSDK provides two basic *CMBReaderDevice* connection layers:

- *MX reader* for barcode scanning with devices like the MX-1000, MX-1100, and MX-1502
- *Camera reader* for barcode scanning with the built-in camera of the mobile device or an MX-100 barcode reader.

Barcode Scanning with an MX Mobile Terminal

The cmbSDK supports Cognex’s MX Series Mobile Terminals with features using cmbSDK:

- **Hardware triggers**: MX Mobile Terminals include two built-in hardware triggers for barcode scanning. MX Mobile Terminals support an optional accessory, the pistol grip with a trigger.
- **Illumination and aiming**: MX Mobile Terminals have built-in illumination and aiming, rendering the live preview on the smartphone's screen unnecessary.
- **Configurations**: Export and import configuration sets to MX Mobile Terminals using Cognex’s DataMan Setup Tool for Windows, the Quick Setup mobile application, or cmbSDK.
- **High-capacity battery**: MX Mobile Terminals have an integrated battery that powers the MX scanning engine and the mobile device. The optional pistol grip includes a second battery that doubles the power capacity of the MX Mobile Terminal.

Getting your MX Mobile Terminal Enabled App into the App Store

NOTE: Before submitting your MX Mobile Termnal Enabled app to the Apple App Store, make sure to add your app to the Cognex MFi product plan. This is a critical step for your app for Apple to approve your app. If your app is not added Apple will reject the app.

Submit a request on https://cmbdn.cognex.com/mfi/apply for each iOS app you want to submit to the App Store.

Once your MFI product plan request was processed, you are notified by e-mail about further steps, at which time you can submit your app to Apple directly.

NOTE: The e-mail notification about the MFI process means that Cognex has placed the request to Apple.It usually takes Apple 3-7 days to process the request.

Update your app’s notes before submitting to the App Store:

1. Log in to iTunes Connect
2. Click on **My Apps**
3. Select the app you would like to submit
4. Click on the app version on the left side of the screen
5. Scroll down to **App Review Information**
6. Update **Notes** with:
    - The related product plan is:
    - Accessory Name: DataMan 9050
    - Product Plan ID: 144826-0004
    - Status: Active Type: Manufacturing Process
    - Phase: Production
7. Click **Save**
8. After completing all changes, click the **Submit for Review** button at the top of the App version information page

Debugging on MX Mobile Terminal

Connect your mobile device (phone or tablet) to your PC via the USB or lightning port to start debugging. If an MX Mobile Terminal is attached to your mobile device via the USB or lightning port while your application is running, you need to debug your application via Wi-Fi.

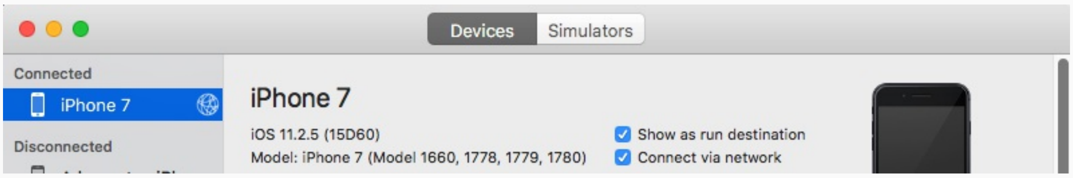
Debugging on iPhone using XCode:

Prerequisites:

- XCode 9 or newer
- iPhone running iOS 11 or newer

If you are running your application with XCode, make sure your device is plugged in via lightning cable and enable *Connect via network* on your mobile device:

1. Open XCode and choose *Window* > *Devices and Simulators* from the top menu.
2. Select your device from the connected list of devices on the left side and check the *Connect via network* checkbox.



iOS 11.2.5 (15D60)
Model: iPhone 7 (Model 1660, 1778, 1779, 1780)
☑ Show as run destination
☑ Connect via network

Now you can close the **Devices** window and start debugging your application without using the lightning cable or USB.

Debugging on iPhone using Xamarin or Visual Studio:

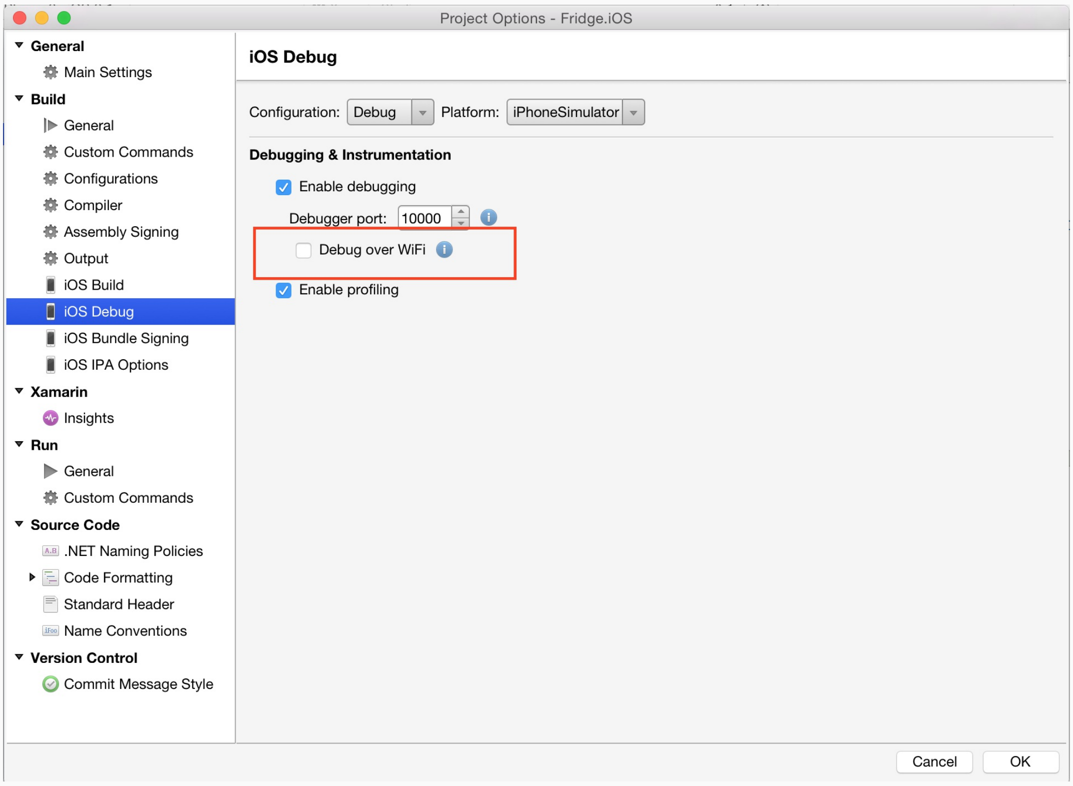
Prerequisites.

- Xamarin
- Visual Studio
- iPhone running iOS 11 or newer

1. Make sure your iPhone is connected using the lightning cable and open your Xamarin.IOS project.
2. Choose *Options* by right-clicking on the project file.
3. Navigate to *iOS Debug* from the left menu and check the *Debug over WiFi* checkbox.

> NOTE: Launch the application through the USB or lightning cable initially.

After launching the application, you can safely unplug and continue your debugging session over Wi-Fi.



iOS Debug

Configuration: Debug Platform: iPhoneSimulator

Debugging & Instrumentation

☑ Enable debugging

Debugger port: 10000

☐ Debug over WiFi

☑ Enable profiling

Barcode Scanning with a Smartphone

Barcode Scanning with a Smartphone or Tablet

The differences in the barcode scanning capabilities of smartphones and purpose-built scanners result in different user experience, impacting the design of the mobile barcode scanning application. By following a few simple guidelines, you can develop applications with the cmbSDK that work the same way when using an MX Mobile Terminal or the built-in camera of a mobile device. Here are some links to start from:

- To initiate barcode scanning without a dedicated hardware trigger, see Mobile Device Triggering.
- To aim for barcode scanning with a smartphone that does not have an aimer, see Mobile Device Aiming.
- To choose the most suitable orientation for barcode scanning, see Mobile Device Orientation.
- To reduce the CPU usage of the mobile device when it performs image analysis and barcode decoding, see Optimizing Mobile Device Performance.

Mobile Device Triggering

Without a hardware trigger, mobile devices must use alternative methods to initiate barcode scanning. The cmbSDK supports three methods to trigger barcode scanning:

- **Application or workflow driven trigger:** The application code or the business logic/workflow of the application invokes the scanning module by calling the *startScanner()* function.
- **Virtual trigger:** To start or stop the scanning process, the application provides a virtual button on the screen. Depending on the application design, you need to press and hold the virtual button to keep the scanner running invoking the scanning module.
- **Simulated trigger:** Press one of the volume control buttons to start or stop the scanning process just like when you pull a trigger on a purpose-built scanner.

Mobile Device Aiming

The built-in camera provides a live-stream preview on the display of the mobile device that can be in partial or full screen, in portrait or landscape orientation, for barcode aiming. Reposition the mobile device until the barcode appears in the field of view of the built-in camera and the application decodes it.

The cmbSDK supports passive aimers: devices attached to the mobile device or its case that use the LED flash of the device as a light source to project an aiming or targeting pattern. The mobile device can project an aimer pattern similar to a purpose-built scanner so live-preview is not needed.

> NOTE: When using the LED flash as an aimer general scanning illumination is not available.

In addition the CmbSDK supports an active aimer that has its own built-in LEDs for illumination and aiming: the MX-100 Barcode Reader. The MX-100 is a mobile device accessory for iOS smartphones attached to the mobile device with a mobile device case. The built-in LED of the MX-100 projects a green dot to help in reading the barcode.

Mobile Device Orientation

The cmbSDK supports portrait orientation, landscape orientation and auto-rotation for both the presentation of the barcode preview and the scan direction. Mobile devices can scan most barcodes regardless of the orientation of the application and the mobile device.

For better read performance read QR, Data Matrix, and Maxicode in portrait orientation, and long codes like PDF417 in landscape orientation.

Optimizing Mobile Device Performance

The cmbSDK is optimized for mobile environment, but image analysis and barcode decoding are still CPU intensive activities. Since these processes share the CPU of the mobile device with the mobile operating system (OS), services, and other applications, the following processes optimize your barcode scanning application and limit it to only using the features of the cmbSDK that they need.

To optimize your application:

- Enable decoding only for the barcode types the application needs to scan.

> NOTE: The cmbSDK supports the decoding of almost 40 different barcode types and subtypes, enabling all results in low performance and unexpected errors.

- Do not enable certain symbologies and/or advanced features at the same time. PDF417, DataMatrix and Dotcode are the most CPU demanding codes, enabled with advanced features they can slow down decoding time.
- Optimize your camera resolution. By default, the cmbSDK uses HD images for barcode decoding.
- Use an appropriate decoder effort level. The cmbSDK has a configurable effort level that controls how aggressively it performs image analysis. The cmbSDK uses a default value (level 2) that is sufficient for most barcodes. Using a higher level can result in better decoding of poorer quality barcodes, resulting in slower performance.

> NOTE: No barcode symbologies are enabled by default, when the cmbSDK is initialized for use with the mobile device's built-in camera.

Using cmbSDK

Using cmbSDK in XCode

Set up your application to use the iOS cmbSDK:

1. Open XCode and start a new project.

2. Add the following lib and frameworks to your project:

```
* SystemConfiguration.framework
* AVFoundation.framework
* CoreGraphics.framework
* CoreMedia.framework
* CoreVideo.framework
* MediaPlayer.framework
* Security.framework
* AudioToolbox.framework
* cmbSDK.framework
```

3. Go to your project's **Info.plist** file and add the **Privacy - Camera Usage Description** or *NSCameraUsageDescription* to display a message about how your application uses the camera of the user's mobile device.

Creating a Swift Bridging Header

To write your app in Swift, you need a bridging header to use the cmbSDK:

1. Create the header by selecting *File -> New File -> Header File*
2. Name the header file and save it, for example **YourApp-Bridging-Header**
3. Open your project settings, under the **Build Settings** tab search for "Objective-C Bridging Header" and add "**$(PROJECT_DIR)/YourApp/YourApp-Bridging-Header.h**". Replace YourApp with the name of your app, and **YourApp-Bridging-Header.h** with the name of your bridging header.
4. Open your bridging header and import the headers that you would like to use from the cmbSDK. For example, import the following headers for basic functionality:

```
#import "CMBReaderDevice.h"
#import "CMBReadResult.h"
#import "CMBReadResults.h"
```

Writing a Mobile Application

The cmbSDK has been designed to provide a high-level, abstract interface for supported scanning devices. This includes not only the MX series of mobile terminals, but also for applications that intend to use the mobile device camera as the imaging device. The intricacies of communicating with and managing these devices is encapsulated within the SDK itself: leaving the application to just connect to the device of choice, then using it.

The primary interface between your application and a supported barcode scanning device is the *CMBReaderDevice* class. This class represents the abstraction layer to the device itself, handling all communication as well as any necessary hardware management (e.g., for smartphone scanning).

Perform the following steps to use the cmbSDK:

1. Initialize a Reader Device for the type of device you want to use: MX reader or camera reader.
2. Connect the Reader Device.
3. Configure the reader (if necessary).
4. Start scanning.

Initialization, connection, and configuration generally need to be performed only once in your application, except for the following cases:

- An MX reader can become disconnected (times out from disuse, dead battery, etc.). A method has been provided to handle this case, and is discussed in a later section.
- Your application has been designed to allow the user to change devices.The cmbSDK is explicitly designed to support this: your application simply disconnects from the current device and establishes a new connection to a different device. The sample application has been written to explicitly demonstrate this capability, which you get when downloading the CmbSDK.

Setting up an Application to Use cmbSDK for iOS

Perform the following steps to set up and start using cmbSDK:

1. Import the following package members, or the classes you use:
    - Swift
    - Objective-C

```
import cmbSDK
```

```
#import <cmbSDK/cmbSDK.h>
```

2. According to your needs:

   - If you want to show partial camera preview, you need a **View** container, for example a **UIView**
   - If you want to use full screen preview (default) you do not need any additional containers.

     - For example if we want to use partial view in our sample application: add a **UIView** in the Main storyboard with the desired dimensions and constraints, and use it in reader device constructor (*previewView* parameter) when reader device is initialized.

   - If you want to display the last scanned image, add a **UIImageView** for container instead of **UIView** for showing the last frame of a preview or scanning session.

   - If you want to display the scanned result as a text, add **UILabel**.

3. Set up the following interfaces to monitor the connection state of the reader and receive information about the read code:

   - Swift
   - Objective-C

```
// MARK: OBSERVER METHODS

//----------------------------------------------------------------------------
// When an applicaiton is suspended, the connection to the scanning device is
// automatically closed by iOS; thus when we are resumed (become active) we
// have to restore the connection (assuming we had one). This is the observer
// we will use to do this.
//----------------------------------------------------------------------------
@objc func appBecameActive() {
    if readerDevice != nil && readerDevice.availability == CMBReaderAvailibilityAvailable && re
aderDevice.connectionState != CMBConnectionStateConnecting && readerDevice.connectionState != C
MBConnectionStateConnected {
        readerDevice.connect(completion: { error in
            if error != nil {
                // handle connection error
            }
        })
    }
}
```

```
// MARK: VIEWCONTROLLER METHODS

override func viewDidLoad() {
    super.viewDidLoad()
    // Add our observer for when the app becomes active (to reconnect if necessary)
    NotificationCenter.default.addObserver(self, selector: #selector(self.appBecameActive), name:UIApplication.didBecomeActiveNotification, object: nil)
}

// MARK: MX Delegate methods

// This is called when a MX-1xxx device has became available (USB cable was plugged, or MX device was turned on),
// or when a MX-1xxx that was previously available has become unavailable (USB cable was unplugged, turned off due to inactivity or battery drained)
func availabilityDidChange(ofReader reader: CMBReaderDevice) {
    self.clearResult()

    if (reader.availability != CMBReaderAvailibilityAvailable) {
        showAlert(title: nil, message: "Device became unavailable")
    } else if (reader.availability == CMBReaderAvailibilityAvailable) {
        self.connectToReaderDevice()
    }
}

// This is called when a connection with the self.readerDevice has been changed.
// The self.readerDevice is usable only in the "CMBConnectionStateConnected" state
func connectionStateDidChange(ofReader reader: CMBReaderDevice) {
    self.isScanning = false
    self.clearResult()

    if self.readerDevice.connectionState == CMBConnectionStateConnected {
        // We just connected, so now configure the device how we want it
        self.configureReaderDevice()
    }

    self.updateUIByConnectionState()
}

// This is called after scanning has completed, either by detecting a barcode, canceling the scan by using the on-screen button or a hardware trigger button, or if the scanning timed-out
func didReceiveReadResult(fromReader reader: CMBReaderDevice, results readResults: CMBReadResults!) {
    self.isScanning = false
    self.btnScan.isSelected = false

    if (readResults.subReadResults != nil) && readResults.subReadResults.count > 0 {
        scanResults = readResults.subReadResults as! [CMBReadResult]
        self.tvResults.reloadData()
    } else if readResults.readResults.count > 0 {
        scanResults = [readResults.readResults.first as! CMBReadResult]
        self.tvResults.reloadData()
    }
}
```

```
#pragma mark OBSERVER METHODS
```

```
//------------------------------------------------------------------------------
// When an applicaiton is suspended, the connection to the scanning device is
// automatically closed by iOS; thus when we are resumed (become active) we
// have to restore the connection (assuming we had one). This is the observer
// we will use to do this.
//------------------------------------------------------------------------------
-(void)appBecameActive {

    if (self.readerDevice != nil &&
        self.readerDevice.availability == CMBReaderAvailibilityAvailable &&
        self.readerDevice.connectionState != CMBConnectionStateConnecting &&
        self.readerDevice.connectionState != CMBConnectionStateConnected)
    {
        [self.readerDevice connectWithCompletion:^(NSError *error) {
            if (error) {
                // handle connection error
            }
        }];
    }
}

#pragma mark VIEWCONTROLLER METHODS

-(void)viewDidLoad {
    [super viewDidLoad];

    // Add our observer for when the app becomes active (to reconnect if necessary)
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(appBecameActive)
                                                 name:UIApplicationDidBecomeActiveNotification
object:nil];
}

#pragma mark MX Delegate methods

// This is called when a MX-1xxx device has became available (USB cable was plugged, or MX devi
ce was turned on),
// or when a MX-1xxx that was previously available has become unavailable (USB cable was unplug
ged, turned off due to inactivity or battery drained)
- (void)availabilityDidChangeOfReader:(CMBReaderDevice *)reader
{
    [self clearResult];

    if (reader.availability != CMBReaderAvailibilityAvailable)
    {
        [self showAlertWithTitle:@"Device became unavailable" message:nil];
    }
    else if (self.readerDevice.availability == CMBReaderAvailibilityAvailable) {
        [self connectToReaderDevice];
    }
}

// This is called when a connection with the self.readerDevice has been changed.
// The self.readerDevice is usable only in the "CMBConnectionStateConnected" state
- (void)connectionStateDidChangeOfReader:(CMBReaderDevice *)reader
{
    self.isScanning = NO;
    [self clearResult];

    if (self.readerDevice.connectionState == CMBConnectionStateConnected){
        // We just connected, so now configure the device how we want it
        [self configureReaderDevice];
    }

    [self updateUIByConnectionState];
}

// This is called after scanning has completed, either by detecting a barcode, canceling the scan by using the on-screen button or a hardware trigger button, or if the scanning timed-out
- (void)didReceiveReadResultFromReader:(CMBReaderDevice *)reader results:(CMBReadResults *)readResults
{
    self.isScanning = false;
    [self.btnScan setSelected:self.isScanning];

    if (readResults.subReadResults && readResults.subReadResults.count > 0) {
        scanResults = readResults.subReadResults;
        [self.tvResults reloadData];
    } else if (readResults.readResults.count > 0){
        CMBReadResult *result = readResults.readResults.firstObject;
        scanResults = @[result];
        [self.tvResults reloadData];
    }
}
```

4. Instantiate a *CMBReaderDevice* object.

Using the MX Reader

Initializing the *CMBReaderDevice* for use with an MX mobile terminal like the MX-1000, MX-1100, or MX-1502 is easy: simply create the reader device using the MX device method (it requires no parameters), and set the appropriate delegate (normally self):

- Swift
- Objective-C

```
let readerDevice:CMBReaderDevice = CMBReaderDevice.readerOfMX()
readerDevice.delegate = self
```

```
CMBReaderDevice *readerDevice = [CMBReaderDevice readerOfMXDevice];
readerDevice.delegate = self;
```

The availability of the MX mobile terminal can change when the device turns ON or OFF, or if the lightning cable gets connected or disconnected. You can handle those changes using the following **CMBReaderDeviceDelegate** method.

- Swift
- Objective-C

```
func availabilityDidChange(ofReader reader: CMBReaderDevice)
```

```
- (void)availabilityDidChangeOfReader:(CMBReaderDevice *)reader
```

Using other Cognex network device

If you want to connect to a Cognex device on the network (e.g handheld or fixed mount) you have to use another framework named as *NetworkDiscovery*. This framework can be found in the cmbSDK bundle.

Starting from 2.3.0 you have to add CocoaAsyncSocket only if you are using *NetworkDiscovery* framework.

Using the Camera Reader or MX-100 Barcode Scanner

Barcode scanning with the built-in camera of the mobile device can be more complex than with an MX mobile terminal. Therefore the cmbSDK supports several configurations to provide the maximum flexibility, including support of optional external aimers and illumination, as well as the ability to customize the appearance of the live-stream preview. MX-100 is such an external device for your iPhone that we call active aimer.

To scan barcodes using MX-100 or the built-in camera of the mobile device, initialize the *CMBReaderDevice* object using the **readerOfDeviceCameraWithCameraMode** static method. The camera reader has several options when initialized. The following parameters are required:

```
* CDMCameraMode
* CDMPreviewOption
* UIView
```

The *CameraMode* parameter is of the type *CDMCameraMode* (defined in **CDMDataManSystem.h**), and it accepts one of the following values:

- **kCDMCameraModeNoAimer**: If no aiming accessory is available, this mode initializes the live-stream preview on the screen to help positioning the barcode in the field of view for detection and decoding.

- **kCDMCameraModePassiveAimer**: Initializes passive aimer use, which is an external accessory that uses the device's built-in LED flash for illumination to project an aiming pattern. In this mode no live-stream preview is presented on the screen.
- **kCDMCameraModeActiveAimer**: Initializes active aimer use, such as the MX-100. Such an attachment has built-in LEDs for projecting an aiming pattern and illumination powered by a built-in battery. In this mode no live-stream preview is presented on the screen.
- **kCDMCameraModeFrontCamera**: Initializes use of the front facing camera. In this mode, illumination is not available.

> NOTE: Front-facing cameras do not have auto focus and illumination as a rule, and provide significantly lower resolution images. This option should be used with care.

The above modes provide the following default settings for the mobile device as a code reader:

- The simulated hardware trigger is disabled.
- When **startScanning()** is called, the decoding process is started. (Seek *CDMPreviewOptionPaused* for more details).

Based on the selected mode, the following additional options and behaviors are set:

- **kCDMCameraModeNoAimer** (NoAimer)
    - The live-stream preview is displayed when the **startScanning()** method is called.
    - Illumination and control button are available and visible on the live-stream preview.
    - Aimer control commands are ignored.

- **kCDMCameraModePassiveAimer** (PassiveAimer)
    - The live-stream preview will not be displayed when the **startScanning()** method is called by default.
    - Illumination is not available
    - Illumination control commands are ignored.

- **kCDMCameraModeActiveAimer** (MX-100)
    - The live-stream preview will not be displayed when the **startScanning()** method is called by default.

  - Illumination is available, if a preview option for camera preview is enabled, the illumination control button is available too.
  - Illumination or aimer control commands are accepted.

- **kCDMCameraModeFrontCamera** (FrontCamera)

  - The live-stream preview is displayed when the **startScanning()** method is called.
  - The front camera is used.
  - Illumination and the control button are not available.
  - Illumination or aimer control commands are ignored.

The *previewOptions* parameter (of type *CDMPreviewOption*, defined in **CDMDataManSystem.h**) is used to change the reader's default values or override defaults derived from the selected **CameraMode**. Multiple options can be specified by OR-ing them when passing the parameter. The available options are the following:

- **kCDMPreviewOptionDefaults**: Accept all defaults set by the **CameraMode**.
- **kCDMPreviewOptionNoZoomBtn**: Hide the zoom button on the live-stream preview.
- **kCDMPreviewOptionNoIllumBtn**: Hide the illumination button on the live-stream preview.
- **kCDMPreviewOptionHwTrigger**: Enable simulated hardware trigger (volume controls) for starting scanning. When pressed, scanning starts.
- **kCDMPreviewOptionPaused**: Display the live-preview when the **startScanning()** method is called without starting the decoding (i.e. looking for barcodes). Pressing the on-screen scanning button starts the decoding.
- **kCDMPreviewOptionAlwaysShow**: Force display of live-preview when active or passive aiming mode has been selected (e.g. *CameraMode == kCDMCameraModePassiveAimer*)
- **kCDMPreviewOptionPessimisticCaching**: Use only when *CameraMode == kCDMCameraModeActiveAimer*, this will read the settings from the **ActiveAimer** when the app resumes from background, in case the aimer settings were changed from another app.

- **kCDMPreviewOptionHighResolution**: Use the device camera in higher resolution to help with scanning small barcodes, but slow decode time. The option sets resolution to 1920x1080 on devices that support it, and the default one on devices that do not.The default resolution is 1280x720.

- **kCDMPreviewOptionHighFrameRate**: Sets the camera to 60 FPS instead of the default 30 FPS to provide a smoother camera preview.

NOTE: The last parameter of type *UIView* is optional and is used as a container for the camera preview. If the parameter is left nil, a full screen preview will be used.

Examples:

Create a reader with no aimer and a full screen live-stream preview:

- Swift
- Objective-C

```
let readerDevice:CMBReaderDevice = CMBReaderDevice.readerOfDeviceCamera(with: CDMCameraMode.noAimer,
 previewOptions:CDMPreviewOption.init(rawValue: 0), previewView:nil)
readerDevice.delegate = self
```

```
CMBReaderDevice *readerDevice = [CMBReaderDevice readerOfDeviceCameraWithCameraMode:kCDMCameraModeNo
Aimer previewOptions:kCDMPreviewOptionDefaults previewView:nil];
readerDevice.delegate = self;
```

Create a reader with no aimer, no zoom button, and using a simulated trigger:

- Swift
- Objective-C

```
let readerDevice:CMBReaderDevice = CMBReaderDevice.readerOfDeviceCamera(with: CDMCameraMode.noAimer,
 previewOptions:[CDMPreviewOption.noZoomBtn, CDMPreviewOption.hwTrigger], previewView:nil)
readerDevice.delegate = self
```

```
CMBReaderDevice *readerDevice = [CMBReaderDevice readerOfDeviceCameraWithCameraMode:kCDMCameraModeNo
Aimer previewOptions:(kCDMPreviewOptionNoZoomBtn | kCDMPreviewOptionHwTrigger) previewView:nil];
readerDevice.delegate = self;
```

Connecting to the Device

Initialize the *CMBReaderDevice* and set a delegate to handle responses from the reader.

Then connect using **connectWithCompletion**:

- Swift
- Objective-C

```
// Make sure the device is turned ON and ready
if self.readerDevice.availability == CMBReaderAvailibilityAvailable {
    // create the connection between the readerDevice object and device
    self.readerDevice.connect(completion: { (error:Error?) in
        if error != nil {
            // handle connection error
        }
    })
}
```

```
// Make sure the device is turned ON and ready
if (readerDevice.availability == CMBReaderAvailibilityAvailable) {
    // create the connection between the readerDevice object and device
    [readerDevice connectWithCompletion:^(NSError *error) {
        if (error) {
            // handle connection error
        }
    }];
}
```

When connected *connectionStateDidChangeOfReader* in the delegate is called, where you can check the connection status in your Reader Device's *connectionState* parameter. It should be **CMBConnectionStateConnected**, which means that you have successfully made the connection to the *CMBReaderDevice*, and can begin using the Cognex Mobile Barcode SDK.

Configuring the Device

To change some settings after connecting to the device the cmbSDK provides a set of high-level, device independent APIs for setting and retrieving the current configuration of the device.

The differences between using an MX reader and the camera reader for scanning are detailed in the following sections.

Configuring MX Mobile Terminals

The MX family of mobile terminals provides sophisticated device configuration and management including saved configurations on the device. MX devices come Cognex preconfigured with most symbologies and features ready to use.

If you would like a custom configuration, reconfigure through DataMan Setup Tool, or the Cognex Quick Setup. Both tools distribute saved configurations easily to multiple devices for simple configuration management.

The mobile application is able to configure the MX device giving you the option to:

- have multiple scanning applications, each of which requiring a different set of device settings
- create your own options in a “known” state, even though the device has been pre-configured correctly

Built-in Camera

The cmbSDK employs a default set of options for barcode reading with the built-in camera of the mobile device. However, there are two important differences to keep in mind:

- The cmbSDK does not implement saved configurations for the built-in camera reader. Every time an application using the camera reader starts defaults ar used automatically.
- The cmbSDK does not enable symbologies by default. The application programmer enables all barcode symbologies to scan in your application. The requisite for enabling only the needed symbologies explicitly, the application achieves most optimal scanning performance on the mobile device.

MX-100

MX-100 is a device-case attachment for iPhones only that provides additional functionalities to the built-in camera such as aiming capabilities and better illumination control. Being a hybrid of an MX device and a built in scanner, the MX-100 has settings for aimer intensity, illumination intensity, and aimer modulation stored on the device, while the rest of the settings, like symbologies settings, are stored in the cmbSDK. See the ***MX-100 User Guide*** for more information.

Here are a few things to keep in mind when using an MX-100 device:

- The MX-100 does not require a license to use the device camera, optionally a free licence can be generated for tracking purposes.
- MX-100 comes pre-configured and the cmbSDK has the following symbologies enabled by default:
    - Code 39
    - Code 128
    - Databar
    - PDF417
    - QR
    - UPC/EAN
- The cmbSDK is extended with a cache mechanism to strengthen optical communication with MX-100. The cache stores all MX-100 settings and it is transparent and available in cmbSDK. Initializing and updating of the cache is the responsibility of

cmbSDK. There are different caches for different settings:

- *Persistent cache*: Settings/values that rarely change (if at all) and SDK can cache on the iPhone for an extended period of time. These items are the MX-100 Serial number, model number, and firmware version. The persistent cache is updated in every 7 days.
- *Session cache*: Settings/values that may change while an application is using an MX-100 (not likely), but should be read from the MX-100 on SDK load/initial connection to the MX-100. These items are: Aimer intensity, Aimer modulation, Aimer timeout, Illumination intensity, and Illumination state.
  By default, the session cache will be maintained optimistically for the best performance. The SDK assumes that another application is not changing the settings of the aimer, the SDK only needs to read the aimer's settings one time, when the initial connection is established.

> NOTE: If another application changes the aimer settings the cache may become out of sync with the aimer. In such a case the cmbSDK gives the possibility to handle the Session cache *pessimistically* where the aimer's configuration is loaded each time the application is resumed. This behavior is accomplished by adding an option flag to the camera connector: **kCDMPreviewOptionPessimisticCaching**.

Enabling Symbologies

Individual symbologies can be enabled using the following method of the *CMBReaderDevice* object:

```
-(void) setSymbology:(CMBSymbology)symbology
enabled:(bool)enabled
completion:(void (^)(NSError *error))completionBlock;
```

All symbologies used for the symbology parameter in this method can be found in **CMBReaderDevice.h**.

Examples

- Swift
- Objective-C

```
self.readerDevice.setSymbology(CMBSymbologyQR, enabled: true, completion: {(_ error: Error?) -> Void
 in
    if error != nil {
        // Failed to enable that symbology, Possible causes are: reader disconnected, out of battery
 or cable unplugged, or symbology not supported by the current readerDevice
    }
})
```

```
[readerDevice setSymbology:CMBSymbologyQR enabled:YES completion:^(NSError *error){
    if (error) {
        // Failed to enable that symbology, Possible causes are: reader disconnected, out of battery
 or cable unplugged, or symbology not supported by the current readerDevice
    }
}];
```

The same method can also be used to turn symbologies off:

- Swift
- Objective-C

```
self.readerDevice.setSymbology(CMBSymbologyUpcEan, enabled: false, completion: {(_ error: Error?) ->
 Void in
    if error != nil {
        // Failed to enable that symbology, Possible causes are: reader disconnected, out of battery
 or cable unplugged, or symbology not supported by the current readerDevice
    }
})
```

```
[readerDevice setSymbology:CMBSymbologyUpcEan enabled:NO completion:^(NSError *error){
    if (error) {
        // Failed to enable that symbology, Possible causes are: reader disconnected, out of battery
 or cable unplugged, or symbology not supported by the current readerDevice
    }
}];
```

Illumination Control

If your reader device is equipped with illumination (e.g. LEDs), you can control whether they are ON or OFF when scanning starts using the following method of your *CMBReaderDevice* object:

- Swift
- Objective-C

```
self.readerDevice.setLightsON(true) { (error:Error?) in
    if error != nil {
        // Failed to enable illumination, Possible causes are: reader disconnected, out of battery o
r cable unplugged, or device doesn't come with illumination lights
    }
}
```

```
[readerDevice setLightsON:YES completion:^(NSError *error) {
    if (error) {
        // Failed to enable illumination, Possible causes are: reader disconnected, out of battery o
r cable unplugged, or device doesn't come with illumination lights
    }
}];
```

Keep in mind that not all devices and device modes supported by the cmbSDK allow illumination control. For example, if using the built-in camera in passive aimer mode, illumination is not available since the LED is being used for aiming.

Camera Zoom Settings

If built-in camera is used as reader device you have the possibility to configure zoom levels and define the way these zoom levels are used.

There are 3 zoom levels for the phone camera, which are:

- normal: not zoomed (100%)
- level 1 zoom (default 200%)
- level 2 zoom (default 400%)

You can define these zoom levels with "SET CAMERA.ZOOM-PERCENT [100-MAX] [100-MAX]" command. It configures how far the two levels will zoom in percentage. 100 is without zoom, and MAX (goes up to 1000) will zoom as far as the device is capable of. First argument is used for setting level 1 zoom, and the second for level 2 zoom.

When you want to check current setting, you can do this with the "GET CAMERA.ZOOM-PERCENT" that returns two values: level 1 and level 2 zoom.

Example

- Swift
- Objective-C

```
readerDevice.dataManSystem()?.sendCommand("SET CAMERA.ZOOM-PERCENT 250 500")
```

```
[readerDevice.dataManSystem sendCommand:@"SET CAMERA.ZOOM-PERCENT 250 500"];
```

Note: Camera needs to be started within SDK at least once to have a valid maximum zoom level. It means that if you set the zoom level to 1000 and the device can go up to 600 only, "GET CAMERA.ZOOM-PERCENT" command returns 1000 as long as camera is not opened (e.g. with [readerDevice startScanning];), but it returns 600 afterwards.

here is another command that sets which zoom level you want to use or returns the actual setting: "GET/SET CAMERA.ZOOM 0-2".

Possible values for the SET command are:

- 0 - normal (un-zoomed)
- 1 - zoom at level 1
- 2 - zoom at level 2

You can call this command before scanning or even during scanning, the zoom goes up to the level that was configured.

When the scanning is finished, the values are reset to normal(0).

Example

- Swift
- Objective-C

```
readerDevice.dataManSystem()?.sendCommand("SET CAMERA.ZOOM 2")
```

```
[readerDevice.dataManSystem sendCommand:@"SET CAMERA.ZOOM 2"];
```

Camera Overlay Customization

When using the built-in camrea of the mobile device, the cmbSDK allows you to see the Camera Preview inside a preview container or in full screen. This preview also contains an overlay, which can be customized. The cmbSDK camera overlay is built from buttons for zoom, flash, closing the scanner (in full screen), a progress bar indicating the scan timeout, and lines on the corners of the camera preview. There are two available overlays: legacy and CMB overlay.

To use the legacy camera overlay, which was used in the cmbSDK v2.0.x and the ManateeWorks SDK, use this property from MWOverlay before initializing the *CMBReaderDevice*:

> NOTE: The legacy overlay has limited customization options, so it is preferred to use the CMB overlay.

- Swift
- Objective-C

```
MWOverlay.setOverlayMode(Int32(OM_LEGACY.rawValue))
```

```
[MWOverlay setOverlayMode:OM_LEGACY];
```

If using the CMB overlay, you can find the layout files in the Resources/layout directory:

CMBScannerPartialView.xib used when the scanner is started inside a container (partial view)

CMBScannerView.xib when the scanner is started in full screen

Copy the layout file that you need, or both layouts, then modify them as you like. Change the size, position or color of the views, remove views, and add your own views, like an overlay image. The views that are used by the cmbSDK (zoom, flash, close buttons, the view used for drawing lines on the corners, and the progress bar) are accessed by the sdk using the *Tag* attribute, make sure the *Tag* attribute remains unchanged, so that the cmbSDK is able to recognize the views and continue to function correctly.

Both the CMB and the legacy overlay allow you to change the images used on the zoom and flash buttons. To do that, first copy the assets folder **MWBScannerImages.xcassets** from the Resources dir into your project. In XCode you can look at the images contained in this assets folder, and replace them with your own while keeping the image names unchanged.

Both the CMB and the LEGACY overlay allow you to change the color and width of the rectangle that is displayed when a barcode is detected. Here's an example on how to do that:

- Swift
- Objective-C

```
MWOverlay.setLocationLineUIColor(UIColor.yellow)
MWOverlay.setLocationLineWidth(5)
```

```
[MWOverlay setLocationLineUIColor:UIColor.yellowColor];
[MWOverlay setLocationLineWidth:5];
```

Advanced Configuration

Every Cognex scanning device implements DataMan Control Commands (DMCC), a method for configuring and controlling the device. Virtually every feature of the device can be controlled using this text based language. The API provides a method for sending DMCC commands to the device. Commands exist both for setting and querying configuration properties.

Appendix A includes the complete DMCC reference for use with the camera reader. DMCC commands for other supported devices (e.g. the MX-1000) are included with the documentation of that particular device.
Appendix B provides the default values for the camera reader's configuration settings as related to the corresponding DMCC setting.
The following examples show different DMCC commands being sent to the device for more advanced configuration.

Example:

Change the scan direction to omnidirectional:

- Swift
- Objective-C

```
self.readerDevice.dataManSystem()?.sendCommand("SET DECODER.1D-SYMBOLORIENTATION 0", withCallback: {
 (response:CDMResponse?) in
    if response?.status == DMCC_STATUS_NO_ERROR {
        // Command was executed successfully
    } else {
        // Command failed, handle errors here
    }
})
```

```
[readerDevice.dataManSystem sendCommand:@"SET DECODER.1D-SYMBOLORIENTATION 0" withCallback:^(CDMResp
onse *response){
    if (response.status == DMCC_STATUS_NO_ERROR) {
        // Command was executed successfully
    } else {
        // Command failed, handle errors here
    }
}];
```

Change the scanning timeout of the live-stream preview to 10 seconds:

- Swift
- Objective-C

```
self.readerDevice.dataManSystem()?.sendCommand("SET DECODER.MAX-SCAN-TIMEOUT 10", withCallback: { (response:CDMResponse?) in
    if response?.status == DMCC_STATUS_NO_ERROR {
        // Command was executed successfully
    } else {
        // Command failed, handle errors here
    }
})
```

```
[readerDevice.dataManSystem sendCommand:@"SET DECODER.MAX-SCAN-TIMEOUT 10" withCallback:^(CDMResponse *response){
    if (response.status == DMCC_STATUS_NO_ERROR) {
        // Command was executed successfully
    } else {
        // Command failed, handle errors here
    }
}];
```

Get the type of the connected device:

- Swift
- Objective-C

```
self.readerDevice.dataManSystem()?.sendCommand("GET DEVICE.TYPE", withCallback: { (response:CDMResponse?) in
    if response?.status == DMCC_STATUS_NO_ERROR {
        // Command was executed successfully
        let deviceType:String = response?.payload
    } else {
        // Command failed, handle errors here
    }
})
```

```
[readerDevice.dataManSystem sendCommand:@"GET DEVICE.TYPE" withCallback:^(CDMResponse *response){
    if (response.status == DMCC_STATUS_NO_ERROR) {
        // Command was executed successfully
        NSString *deviceType = response.payload;
    } else {
        // Command failed, handle errors here
    }
}];
```

Resetting the Configuration

NOTE: This section only contains instruction to reset cmbSDK defaults. For information on resetting to factory defaults please refer to the manual of the reader device.

The cmbSDK includes a method for resetting the device to its default settings. In the case of an MX mobile terminal, this is the configuration saved by default, while in the case of the built-in camera, these are the defaults identified in Appendix B, where no symbologies will be enabled. This method is the following:

- Swift
- Objective-C

```
self.readerDevice.resetConfig { (error:Error?) in
    if error != nil {
        // Failed to reset configuration, Possible causes are: reader disconnected, out of battery o
r cable unplugged
    }
}
```

```
[readerDevice resetConfigWithCompletion:^(NSError *error) {
    if (error) {
        // Failed to reset configuration, Possible causes are: reader disconnected, out of battery o
r cable unplugged
    }
}];
```

Scanning Barcodes

With a properly configured reader, you are ready to scan barcodes. This is simply accomplished by calling the **startScanning()** method from your *CMBReaderDevice* object. What happens next is based on the type of *CMBReaderDevice* and how it has been configured. Generally:

- If using an MX terminal, press a trigger button on the device to turn the scanner on and read a barcode.
- If using the camera reader, the cmbSDK starts the camera, displays the configured live-stream preview, and begins analyzing the frames from the video stream, looking for a configured barcode symbology.

Scanning stops under one of the following conditions:

- The reader found and decoded a barcode.
- The user released the trigger or pressed the stop button on the live-stream preview screen.
- The camera reader timed out without finding a barcode.
- The application program calls the **stopScanning()** method.

When a barcode is decoded successfully, you will receive a *CMBReadResults* array in your *CMBReaderDevice*'s delegate using the following **CMBReaderDeviceDelegate** method:

- Swift
- Objective-C

```
func didReceiveReadResult(fromReader reader: CMBReaderDevice, results readResults: CMBReadResults!)
```

```
- (void)didReceiveReadResultFromReader:(CMBReaderDevice *)reader results:(CMBReadResults *)readResults;
```

To simply display a *ReadResult* after scanning a barcode:

- Swift
- Objective-C

```
func didReceiveReadResult(fromReader reader: CMBReaderDevice, results readResults: CMBReadResults!)
{
    if readResults.readResults.count > 0 {
        let readResult:CMBReadResult = readResults.readResults?.first as! CMBReadResult
        if readResult.image != nil {
            self.ivPreview.image = readResult.image
        }
        if readResult.readString != nil {
            self.lblCode.text = readResult.readString
        }
    }
}
```

```
- (void)didReceiveReadResultFromReader:(CMBReaderDevice *)reader results:(CMBReadResults *)readResults {
    if (readResults.readResults.count > 0) {
        CMBReadResult *readResult = readResults.readResults.firstObject;
        if (readResult.image) {
            self.ivPreview.image = readResult.image;
        }
        if (readResult.readString) {
            self.lblCode.text = readResult.readString;
        }
    }
}
```

In the example above, *ivPreview* is an UIImageView used to display an image of the barcode that was scanned, and *lblCode* is a UILabel used to show the result from the barcode. You can also use the BOOL from *readResult.goodRead* to check whether the scan was successful or not.

Working with Results

When a barcode is successfully read, a *CMBReadResult* object is created and returned by the **didReceiveReadResultFromReader:results:** method. In case of having multiple barcodes successfully read on a single image/frame, multiple *CMBReadResult* objects are returned. This is why the *CMBReadResults* class has an array of *CMBReadResult* objects containing all results.

The *CMBReadResult* class has properties describing the result of a barcode read:

- **goodRead** (BOOL): tells whether the read was successful or not
- **readString** (NSString): the decoded barcode as a string
- **image** (UIImage): the image/frame that the decoder has processed
- **imageGraphics** (NSData): the boundary path of the barcodeas SVG data
- **XML** (NSData): the raw XML that the decoder returned
- **symbology** (CMBSymbology): the symbology type of the barcode. This enum is defined in **CMBReaderDevice.h**.

When a scanning ends with no successful read, a *CMBReadResult* is returned with the **goodRead** property set to false. This usually happens when scanning is canceled or timed out.

To enable the image and **imageGraphics** properties being filled in the *CMBReadResult* object, you have to set the corresponding **imageResultEnabled** and/or

SVGResultEnabled properties of the *CMBReaderDevice* object.

To see an example on how the image and SVG graphics are used and displayed in parallel, refer to the sample applications provided in the SDK package.

To access the raw bytes from the scanned barcode, you can use the XML property. The bytes are stored as a Base64 String under the "full_string" tag. Here's an example how you can use the iOS XML parser to extract the raw bytes from the XML property.

Example:

Parsing the XML and extracting the Base64 String is done using the *XMLParserDelegate* delegate. Add this delegate and the following methods from it in your ViewController:

- Swift
- Objective-C

```
// XMLParserDelegate
var currentElement = ""
var base64String = ""
func parser(_ parser: XMLParser, didStartElement elementName: String, namespaceURI: String?, qualifiedName qName: String?, attributes attributeDict: [String : String] = [:]) {
    currentElement = elementName
}

func parser(_ parser: XMLParser, foundCharacters string: String) {
    if currentElement == "full_string" {
        base64String = string
    }
}
```

```
#pragma NSXMLParserDelegate
NSString *currentElement;
NSString *base64String;
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName attributes:(NSDictionary<NSString *,NSString *> *)attributeDict {
    currentElement = elementName;
}

- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string {
    if ([currentElement isEqualToString:@"full_string"]) {
        base64String = string;
    }
}
```

After you have set the *XMLParserDelegate* to extract the base64 string from the XML result, you need to create a *XMLParser* object and parse the result.xml using this delegate. This can be done when receiving the scan result in the *CMBReaderDeviceDelegate*, or when accessing a *CMBReadResult* object. Here's how you can get the raw bytes using the delegate that you created earlier:

- Swift
- Objective-C

```
let xmlParser:XMLParser = XMLParser.init(data: result.xml)
xmlParser.delegate = self
if xmlParser.parse() {
    // Access the raw bytes via this variable
    let bytes:Data? = Data.init(base64Encoded: base64String)
}
```

```
NSXMLParser *xmlParser = [NSXMLParser.alloc initWithData:result.XML];
xmlParser.delegate = self;
if ([xmlParser parse]) {
    // Access the raw bytes via this variable
    NSData *bytes = [NSData.alloc initWithBase64EncodedString:base64String options:0];
}
```

Image Results

By default, the image and SVG results are disabled, which means that when scanning, the CMBReadResults will not contain any data in the corresponding properties.

> Not all supported devices provide SVG graphics.

To enable image results, set the imageResultEnabled property from the CMBReaderDevice class by using the following method:

- Swift
- Objective-C

```
self.readerDevice.imageResultEnabled = true
```

```
[readerDevice setImageResultEnabled:YES];
```

To enable SVG results, set the imageResultEnabled property from the CMBReaderDevice class by using the following method:

- Swift
- Objective-C

```
self.readerDevice.svgResultEnabled = true
```

```
[readerDevice setSVGResultEnabled:YES];
```

Handling Disconnection

1. Disconnection:

There may be cases when a device disconnects due to low battery condition or manual cable disconnection. These cases can be detected by the *connectionStateDidChangeOfReader* callback of the *CMBReaderDeviceDelegate*.

> **Note:** The **availabilityDidChangeOfReader** method is also called when the device becomes physically unavailable. It means that the (re)connection is not possible. Always check the availability property of the *CMBReaderDevice* object before trying to call the **connectWithCompletion** method.

2. Re-Connection:

After returning to your application from inactive state, the reader device remains initialized but not connected. There is no need for reinitializing the SDK but you need to re-connect.

Some iOS versions will send an "Availability" notification when resuming the application that the external accessory is available. You can use this in the **CMBReaderDeviceDelegate** method: *(void)availabilityDidChangeOfReader: (CMBReaderDevice *)reader so* when the reader becomes available, you can connect.

For example:

- Swift

- Objective-C

```
func availabilityDidChange(ofReader reader: CMBReaderDevice) {
    if (reader.availability == CMBReaderAvailibilityAvailable) {
        readerDevice.connect(completion: { error in
            if error != nil {
                // handle connection error
            }
        })
    }
}
```

```
- (void)availabilityDidChangeOfReader:(CMBReaderDevice *)reader {
    if (readerDevice.availability == CMBReaderAvailibilityAvailable) {
        [readerDevice connectWithCompletion:^(NSError *error) {
            if (error) {
               // handle connection error
            }
        }];
    }
}
```

Some iOS versions do not report availability change on resume, so you need to handle this manually. Add an observer for *UIApplicationDidBecomeActiveNotification* and connect.

> NOTE: Make sure that the reader is not already in "connecting" or "connected" state.

Example:

- Swift
- Objective-C

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Reconnect when app resumes
    NotificationCenter.default.addObserver(self, selector: #selector(self.appBecameActive), name:NSN
otification.Name.UIApplicationDidBecomeActive, object: nil)
}

// handle app resume
func appBecameActive() {
    if readerDevice != nil
        && readerDevice.availability == CMBReaderAvailibilityAvailable
        && readerDevice.connectionState != CMBConnectionStateConnecting && readerDevice.connectionSt
ate != CMBConnectionStateConnected {
        readerDevice.connect(completion: { error in
            if error != nil {
                // handle connection error
            }
        })
    }
}
```

```
- (void)viewDidLoad {
     // Reconnect when app resumes
     [[NSNotificationCenter defaultCenter] addObserver:self
                                                 selector:@selector(appBecameActive)
                                                     name:UIApplicationDidBecomeActiveNotification o
bject:nil];
}

// handle app resume
-(void) appBecameActive {
    if (readerDevice != nil
        && readerDevice.availability == CMBReaderAvailibilityAvailable
        && readerDevice.connectionState != CMBConnectionStateConnecting && readerDevice.connectionSt
ate != CMBConnectionStateConnected) {
        [readerDevice connectWithCompletion:^(NSError *error) {
            if (error) {
                // handle connection error
            }
        }];
    }
}
```

Appendix A - DMCC for the Camera Reader

Appendix A - DMCC for the Camera Reader

The following table lists the various DMCC commands supported by the cmbSDK when using the built-in camera for barcode scanning.

> **Note**: Many of the cmbSDK commands are also supported on the MX mobile terminals and the MX-100. Commands that are supported by the MX Terminal or MX-100 are indicated with an x in the last two columns.

GET/SET	COMMAND	PARAMETER(S)	
GET	BATTERY.CHARGE	[0-100]	Displays curr
	BEEP	repetition [0-3] level [0-2]	Plays t
GET/SET	BEEP.GOOD	number of beeps [0-3] beep tone [0-2]	Sets the number of beeps For the built-in camera, 0 1 turns the b
GET/SET	CAMERA.ZOOM	0-2	Values t 0 1 2 The zoom level is used duri
GET/SET	CAMERA.ZOOM-PERCENT	Level 1 [100-MAX] Level 2 [100-MAX]	Gets or sets t le le Note: Make sure to start th have a proper value for ma
GET/SET	CODABAR.CODESIZE	any min max	[ON | OFF] [1-max] Sets [min-80] Sets
GET/SET	C11.CHKCHAR	[ON | OFF]	Turns
GET/SET	C11.CHKCHAR-OPTION	[0-1]	0: disable 1: enable
GET/SET	C11.CODESIZE	any min	[ON|OFF] [1-max] Sets

		max	[min-80] Sets
GET/SET	C25.CODESIZE	any min max	Code 25 Code Size. For the same minimum length; [ON|OFF] [1-max] Sets [min-80] Sets
GET/SET	C39.ASCII	[ON|OFF]	Turns Cod
GET/SET	C39.CODESIZE	any min max	[ON|OFF] [2|4-max] Sets min length using camera API, [min-50|80] Sets max leng for using camera AP Setting codesize to 2
GET/SET	C39.CHKCHAR	[ON|OFF]	Cod
GET/SET	C93.ASCII	[ON|OFF]	Turns Cod
GET/SET	C93.CODESIZE	any min max	[ON|OFF] [1-max] Sets [min-80] Sets
	CONFIG.DEFAULT		Resets most of the camera communications settings
	CONFIG.SAVE		Saves the current configur Note that when an MX pow configuration is re
	CONFIG.RESTORE		Restores the saved config
		0 1	Specifies results to be Text

GET/SET	DATA.RESULT-TYPE	2 4 8	Scan imag
GET/SET	DATABAR.EXPANDED	[ON|OFF]	Turns the Datal
GET/SET	DATABAR.LIMITED	[ON|OFF]	Turns the Dat
GET/SET	DATABAR.GROUP DATABAR.RSS14	[ON|OFF]	Turns the DataBar G RSS
GET/SET	DATABAR.RSS14STACK	[ON|OFF]	Turns the DataBar RSS14 Sta cmbSDK 2.4.
GET/SET	DECODER.1D-SYMBOLORIENTATION	0 1 2 3	Use omni Use horizont Use v Use ho
GET/SET	DECODER.CENTERING-WINDOW	[0-100] [0-100] [0-100] [0-100]	Location and size of centerin Center horizontally Center vertically a Size horizontally Size vertically as
GET/SET	DECODER.DISPLAY-TARGET	[ON|OFF]	Displays
GET/SET	DECODER.EFFORT	[0-5]	Sets the effort l NOTE: Do not use 4-5 for c
GET/SET	DECODER.MAX-SCAN-TIMEOUT	[1-120]	Sets the timeout for the live- decoding is paused, and th
GET	DECODER.MAX-THREADS		Returns the max number
GET/SET	DECODER.REREAD-TIME	[0-10000]	Code re-re

GET/SET	DECODER.ROI-PERCENT	[0-100] [0-100] [0-100] [0-100]	Location and size of regio Center hori Size horizo Center ve Size vert
GET/SET	DECODER.TARGET-DECODING	[ON|OFF]	Enable target decoding, to
GET/SET	DECODER.THREADS-USED	[0-MAX]	Specify the max number of th
GET/SET	DECODER.USE-CENTERING	[ON|OFF]	Only reads co
	DEVICE.DEFAULT		Resets the c
GET	DEVICE.FIRMWARE-VER		Gets the
GET	DEVICE.ID	string	Returns device ID assi For a built For MX-100 Ba
GET/SET	DEVICE.NAME		Returns the r
GET	DEVICE.SERIAL-NUMBER		Returns the For a built-in camera, th
GET	DEVICE.TYPE		Returns the device name a For a built-in ca If MX-100 is a
GET/SET	FOCUS.FOCUSTIME	[0-10]	Sets the camera's auto-f a For M
GET/SET	I2O5.CHKCHAR	[ON|OFF]	Turns Interle

GET/SET	I2O5.CODESIZE	any min max	For the cmbSDK, all of th length; thus it will accep [ON|OFF] Acce [1-max] Sets min le [min-80] Sets max
GET/SET	IMAGE.FORMAT	0 1 2	Scanner retur Scanner retu Scanner retu
GET/SET	IMAGE.QUALITY	[10, 15, 20, ...90]	Speci
GET/SET	IMAGE.SIZE	0 1 2 3	Scanne Scanne Scanne Scanner
GET/SET	LIGHT.AIMER	[0-1]	Disables/enables Default 0: No 1: Passi
SET	LIGHT.AIMER-CONFIG	[32-100] [0-15] [32-100]	Sets i
GET	LIGHT.AIMER-CONFIG	[0-1]	Get's all of the MX-10 0: reads t 1: alwa
GET/SET	LIGHT.AIMER-INTENSITY	[32-100]	Sets/gets the aim
GET/SET	LIGHT.AIMER-MODULATION	[0-15]	Sets/gets the aimer LED
GET/SET	LIGHT.AIMER-TIMEOUT	[0-600]	Timeout This value is always ove
GET/SET	LIGHT.INTERNAL-ENABLE	[ON|OFF]	Enable

GET/SET	MSI.CHKCHAR	[ON | OFF]	Turns MSI D
GET/SET	MSI.CHKCHAR-OPTION	0 1 2 3 4 5	Us Use m Use mod 1 Use mod 11 m Use mod 1 Use mod 11 mo
GET/SET	MSI.CODESIZE	mode min max	[ON|OFF] Ac [1-max] Sets min/ [min-80] Sets min/
GET/SET	SYMBOL.AZTECCODE	[ON | OFF]	Turns the A
GET/SET	SYMBOL.CODABAR	[ON | OFF]	Turns the
GET/SET	SYMBOL.C11	[ON | OFF]	Turns the
GET/SET	SYMBOL.C128	[ON | OFF]	Turns the C
GET/SET	SYMBOL.C25	[ON | OFF]	Turns the Code
GET/SET	SYMBOL.C39	[ON | OFF]	Turns the
GET/SET	SYMBOL.C39-CONVERT-TO-C32	[ON | OFF]	Enables/disables th
GET/SET	SYMBOL.C93	[ON | OFF]	Turns the
GET/SET	SYMBOL.COOP	[ON | OFF]	Turns the COOP s

GET/SET	SYMBOL.DATAMATRIX	[ON | OFF]	Turns the D
GET/SET	SYMBOL.DATABAR	[ON | OFF]	Turns the DataBar symbo DATABAR.LIMITED, DATABAR.E D
GET/SET	SYMBOL.DOTCODE	[ON | OFF]	Turns the
GET/SET	SYMBOL.IATA	[ON | OFF]	Turns the IATA sy
GET/SET	SYMBOL.INVERTED	[ON | OFF]	Turns the Inverted
GET/SET	SYMBOL.ITF14	[ON | OFF]	Turns the ITF-14 s
GET/SET	SYMBOL.UPC-EAN	[ON | OFF]	Turns the UPC-A, UPC-E
GET/SET	SYMBOL.MATRIX	[ON | OFF]	Turns the Matrix s
GET/SET	SYMBOL.MAXICODE	[ON | OFF]	Turns the M
GET/SET	SYMBOL.MSI	[ON | OFF]	Turns the M
GET/SET	SYMBOL.PDF417	[ON | OFF]	Turns the
GET/SET	SYMBOL.PLANET	[ON | OFF]	Turns the
GET/SET	SYMBOL.POSTNET	[ON | OFF]	Turns the
GET/SET	SYMBOL.TELEPEN	[ON | OFF]	Turns the

GET/SET	SYMBOL.4STATE-AUS	[ON | OFF]	Turns the Au
GET/SET	SYMBOL.4STATE-IMB	[ON | OFF]	Turns the Intellige
GET/SET	SYMBOL.4STATE-RMC	[ON | OFF]	Turns the Roy
GET/SET	SYMBOL.QR	[ON | OFF]	Turns the QR a
GET/SET	TRIGGER.TYPE	0 1 2 3 4 5	
GET/SET	UPC-EAN.EAN13	[ON | OFF]	Turns the
GET/SET	UPC-EAN.EAN8	[ON | OFF]	Turns the
GET/SET	UPC-EAN.UPC-A	[ON | OFF]	Turns the
GET/SET	UPC-EAN.UPC-E	[ON | OFF]	Turns the
GET/SET	UPC-EAN.UPCE1	[ON | OFF]	Turns the
GET/SET	UPC- EAN.SUPPLEMENT	0 1-4	Turns off UPC supplementa
GET/SET	VIBRATION.GOOD	[ON | OFF]	Sets/gets whether to vi

Appendix B - Camera Reader Defaults

Appendix B - Camera Reader Defaults

The following table lists the defaults the SDK uses on startup for the camera reader.

> **Note:** At the low-level, the cmbSDK supported devices can perform two types of configuration resets: a device reset and a config reset. A device reset restores all configuration properties to their saved defaults, while a config reset restores mostly the scanning settings, leaving communication settings alone. In the table below, those items that are only reset by a device reset are indicated.

> **Note:** The Reader Device method resetConfig() performs a config reset. To perform a device reset, the DMCC command DEVICE.RESET would need to be issued.

SETTING	DEFAULT VALUE	DEVICE RESET ONLY?
BEEP.GOOD	1 1 (Turn beep on)	
C11.CHKCHAR	OFF	
C11.CHKCHAR-OPTION	1	
C39.ASCII	OFF	
C39.CHKCHAR	OFF	
C93.ASCII	OFF	
COM.DMCC-HEADER	1 (Include Result ID)	Y

COM.DMCC-RESPONSE	0 (Extended)	Y
DATA.RESULT-TYPE	1	Y
DECODER.1D-SYMBOLORIENTATION	1	
DECODER.EFFORT	2	
DECODER.MAX-SCAN-TIMEOUT	60	
DEVICE.NAME	“MX-“ + the last six digits of DEVICE.SERIAL-NUMBER	
Symbologies (SYMBOL.*)	OFF (all symbologies are disabled)	
Symbology sub-types (groups): DATABAR.EXPANDED DATABAR.LIMITED DATABAR.RSS14 DATABAR.RSS14STACK UPC-EAN.EAN13 UPC-EAN.EAN8 UPC-EAN.UPC-A UPC-EAN.UPC-E UPC-EAN.UPCE1	ON OFF OFF OFF ON ON ON ON OFF	
FOCUS.FOCUSTIME	3	
I2O5.CHKCHAR	OFF	
IMAGE.FORMAT	1 (JPEG)	
IMAGE.QUALITY	50	
IMAGE.SIZE	1 (1/4 size)	

LIGHT.AIMER	Default based on cameraMode: 0: NoAimer and FrontCamera 1: PassiveAimer and ActiveAimer	Y
LIGHT.AIMER-TIMEOUT	60	
LIGHT.INTERNAL-ENABLE	OFF	

Appendix B - Camera Reader Defaults

Setting	Default Value	Device Reset Only?
Minimum/maximum code lengths	ON 4 40	
MSI.CHKCHAR	OFF	
MSI.CHKCHAR-OPTION	0	
TRIGGER.TYPE	2 (Manual)	
UPC-EAN.SUPPLEMENT	0	

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.

The good news is that the cmbSDK is backward compatible with the MW SDK. The cmbSDK simply adds a higher-level API to the scanning methods that utilize the camera of a smartphone or tablet. Or, you can continue to use the lower-level methods you have become familiar with in the Manatee Works SDK. Your account, login, license(s), and key(s) remain the same. If you do decide to program to the higher-level API, 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 we need to completely remove the MW library.

Please remove the following files from the mwSDK:

- libBarcodeScanner.a
- MWResult.h/m
- MWOverlay.h/m
- MWParser.h/m

Optionally, if you don't use the helper classes MWImageGetter.h/m and MWImageScanner.h/m you can remove them as well.

Add cmbSDK

Next step is to add **cmbSDK framework** and use in your project. Please navigate to this url to check step by step how to integrate cmbSDK inside your project.

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 don't have that **CMBReaderDevice** object and we use API methods from the **BarcodeScanner** class 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 of these things are done in code behind with default values when we create a CMBReaderDevice object. Here some of the settings can be

set in the constructor as input parameter and others can be set/changed after we create and connect to reader device. Using cmbSDK not only creating reader device in enough to start scanning process, we also need to connect to reader device and set necessary delegate methods 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. We do that by creating an AVCaptureDevice object with AVCaptureSession and use that to capture frames from the device's camera, which we then decode with the mwSDK. Using cmbSDK there is only one method to start the scanning process and comes from CMBReaderDevice object (startScanning). We can't start scanning process if we don't have valid connection to reader device. Here we can scan in full screen mode if we create reader device without setting a previewContainer, or if we want to scan in a partial view, we need to create a UIVIew container for the preview that container in our layout and use it as an input parameter. Result from scanning process will be received in didReceiveReadResultFromReader:(CMBReaderDevice *)reader results:(CMBReadResults *)readResults function from the CMBReaderDeviceDelegate delegate.
3. **Result received**
   - If we have a successful read or we stop the scanning process and have no read, result object will be received in the didReceiveReadResultFromReader:(CMBReaderDevice *)reader results:(CMBReadResults *)readResults function. In cmbSDK the result object is more extended than in mwSDK. From that object we can read our barcode result, symbology, image from the last frame, SVG result, etc.