Cognex Mobile Barcode SDK for iOS (v2.1.x)

Introduction

iOS (formerly **iPhone OS**) is a mobile operating system created and developed by Apple Inc. exclusively for its hardware. It is the operating system that presently powers many of the company's mobile devices, including the iPhone, iPad, and iPod Touch. It is the second most popular mobile operating system globally after Android.

The iOS user interface is based upon direct manipulation, using multi-touch gestures. Interface control elements consist of sliders, switches, and buttons. Interaction with the OS includes gestures such as *swipe*, *tap*, *pinch*, and *reverse pinch*, all of which have specific definitions within the context of the iOS operating system and its multi-touch interface. Internal accelerometers are used by some applications to respond to shaking the device (one common result is the undo command) or rotating it in three dimensions (one common result is switching between portrait and landscape mode). Apple has been significantly praised for incorporating thorough accessibility functions into iOS, enabling users with vision and hearing disabilities to properly use its products.

Major versions of iOS are released annually. The current version, iOS 12, was released on October 8, 2018. It is available for the iPhone 5S and later iPhone models, the fifth-generation iPad, the iPad Air and iPad Air 2, the iPad Pro, the iPad Mini 2 and later iPad Mini models, and the sixth-generation iPod Touch. In iOS, there are four abstraction layers: the Core OS, Core Services, Media, and Cocoa Touch layers.

Overview

The Cognex Mobile Barcode SDK (cmbSDK) is a simple, yet powerful tool for developing mobile barcode scanning applications. Based on Cognex's flagship DataMan technology and the Manatee Works Barcode Scanning SDK, the cmbSDK allows developers to create barcode scanning applications for the entire range of mobile scanning devices: from smartphones and tablets to the MX line of high-performance, industrial barcode scanners. By adhering to a few simple guidelines, developers can write applications that will work with any supported MX mobile terminal or smartphone with little or no conditional code. The SDK achieves this by abstracting the device through a "reader" connection layer: once the application establishes its connection with the desired reader, a single, unified API is used to configure and interface with the device.

The SDK provides two basic readers:

- “MX reader” for barcode scanning with devices like the MX-1000 and MX-1502
- “camera reader” for barcode scanning using the built-in camera of the mobile device or strengthen its capabilities with an MX-100 barcode reader.

Legal Notices

Barcode Scanning with an MX Mobile Terminal

The SDK supports Cognex's line of MX mobile terminals, including the MX-1000 and MX-1502 devices. You can get a detailed description of these models at the official website of Cognex (https://www.cognex.com). Some of the relevant features of these devices regarding the cmbSDK usage are the following:

- Hardware trigger: MX mobile terminals include two built-in triggers for barcode scanning, and support an optional pistol grip with trigger.
- Illumination and aiming: MX mobile terminals have built-in illumination and aiming.
- Stored configurations: An MX mobile terminal can be preconfigured using Cognex's DataMan Setup Tool for Windows, the Quick Setup mobile application, or the SDK itself. The MX mobile terminal can save and restore multiple configuration sets.
- High-capacity battery: Each MX mobile terminal has an integrated battery that not only powers the MX scanning engine, but also provides power to the mobile device. The optional pistol grip includes a second battery that doubles the MX's power capacity.

The following features of the MX platform combine to make application development with them straightforward.

- Ease of setup: MX mobile terminals come preconfigured to provide an exceptional out-of-the-box experience. In many cases, it is not even necessary to change the defaults of the device. Also, since the MX mobile terminals have saved configurations that can be distributed to all your devices, setup is usually not necessary at the application level. Nevertheless, it is often desirable to put the device in a "known" state when the barcode scanning application starts, so the cmbSDK provides methods to restore the device defaults as well as to control individual settings.
- Having illumination and aimer, there is no need to have a live preview on the smartphone's screen like traditional, mobile barcode scanning applications often do. MX mobile terminals do not even support a "livestream" decoding mode.

Getting your MX Mobile Terminal Enabled App into the App Store

Before submitting your MX-1000 Enabled app to the Apple App Store, your app must be added to the Cognex MX-1000 MFi product plan. This is a critical step for your app to be approved by Apple. (If your app isn't added to the plan, Apple will reject it.)

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

- Name of app as it will appear in App Store
- App version number
- App Store category
- Bundle identifier
- External Accessory protocols (which must include at least com.cognex.dmcc)
- Brief functional overview of app and its key features
- Name of the developer that will submit the app to the App Store
- Expected release date

You will also need to update your app's notes before submitting to the App Store. Please follow the instructions to do this below:

- Log in to iTunes Connect
- Click on "My Apps"
- Select your app
- Click on the app version on the left side of the screen
- Scroll down to "App Review Information"
- Update "Notes" with:

    The related product plan is:

    Accessory Name: DataMan 9050

    Product Plan ID: 144826-0004

    Status: Active Type: Manufacturing Process

    Phase: Production

- Click "Save"
- Once you've completed all changes, click the "Submit for Review" button at the top of the App version information page.

Once this information has been received, Cognex will add your app to the MX-1000 product plan. You will receive an email confirmation when this step is completed at which time you can submit your app to Apple directly.

Debugging on MX Mobile Terminal

The MX Mobile Terminals connect to your device via the device's usb or lightning port. This means that the port will be occupied while your application is running. There are other ways to debug your application and we will discuss how to debug via wifi below.

Debugging on iPhone using XCode:

Requirements:

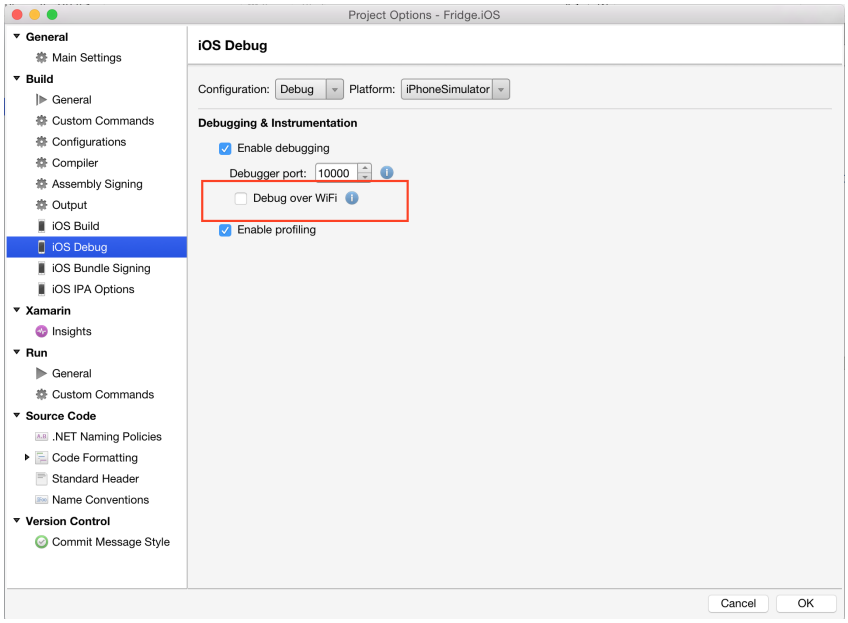
- XCode 9 or newer
- iPhone running iOS 11 or newer

If you are running your application with XCode, you need to enable "Connect via network" on your device. To do that, first open XCode and from the top menu choose "Window" and then "Devices and Simulators". Make sure your device is plugged in via lightning cable at this point. Select your device from the "Connected" list of devices on the left side, and check the "Connect via network" checkbox.

At this point, you can close the Devices window and you can run your application without using the lightning cable.

Debugging on iPhone using Xamarin or Visual Studio:

Make sure your iPhone is connected using the lightning cable and open your Xamarin.IOS project. Go to your project options by right-clicking project and choosing "Options". Navigate to iOS Debug from the left menu, and check the "Debug over WiFi" checkbox. Launching of application is still done through the USB cable, so the initial launch will still require you to have the cable plugged. But once started, you can safely unplug and continue your debugging session over WiFi.



Barcode Scanning with a Smartphone - iOS

Barcode Scanning with a Smartphone

It is important to recognize that there are several fundamental differences in the capabilities of smartphones (and tablets) as barcode scanning devices. These differences result in a user experience different from purpose-built scanners, impacting the design of the mobile barcode scanning application.

These differences and the general impact they have on your application are the following:

- A smartphone does not have a dedicated hardware trigger. Without a hardware trigger, the application program itself is generally responsible for initiating the scanning process, which results in accessing the built-in camera, displaying a preview screen if required, and analyzing captured frames from the video stream for barcodes.
- A smartphone (unless otherwise configured) does not have an aimer. Generally, the application program provides a live-stream camera preview on the mobile device screen, thereby allowing the user to see what the camera sees and can then position the device over the barcode. Adding a purpose built device like MX-100 can help the user to aim at the barcode, without having to look at the screen, or having to use a camera preview at all and can also further improve the scanning process by illuminating the barcodes.
- Mobile device orientation may need to be considered. Most users hold and use a mobile device primarily in a portrait orientation and for barcode scanning. Having the camera in this orientation is generally sufficient. However, most mobile device cameras have a higher resolution along their landscape orientation. When scanning very long or dense barcodes, reorienting the device to landscape can be beneficial and even necessary to decode these barcodes.
- Image analysis and barcode decoding is performed in software on the mobile device which can be a CPU intensive task. For this reason (and others discussed later), it is highly recommended to only enable the symbologies and features of the SDK your application will need, not everything the cmbSDK is capable of.

The cmbSDK has been specifically engineered to make these differences as transparent as possible to the application developer and the user. By following a few simple guidelines, it is possible to develop applications that work and behave the same, whether using an MX-1000 mobile terminal, or just the built-in camera of the device.

Mobile Device Triggering

Without a hardware trigger, mobile devices must use alternative methods to initiate barcode scanning. There are three common paradigms used:

- **Application or workflow driven trigger**: In this paradigm, it is the application code itself, or the business logic/workflow of the application that starts the scanning process. In other words, the user of the application has reached a point where a barcode needs to be scanned, so the application invokes the scanning module. In simple programming terms, this is akin to calling a function like "startScanner()".
- **Virtual trigger**: This is where the application program provides a button on the screen whereby the user can use to start/stop the scanning process. Depending on the application design, the user may be required to press and hold the virtual button to keep the scanner running. This method is similar to the workflow driven method as the button from the user interface is merely being used to invoke the scanning module.
- **Simulated trigger**: For this method, one of the buttons on the mobile device, typically the volume-down button, is used to simulate a hardware trigger. When the user presses and holds this button, the scanner starts/stops just like when a trigger is pulled on a purpose-built scanner. This method is not commonly used as users find it non- intuitive and inconvenient to use the volume key in this fashion.

The cmbSDK supports all three of these methods, any one of which (or multiple) can be used in an application.

Mobile Device Aiming

Mobile devices, not considering purpose built scanners, do not have a built-in aimer. Barcode aiming is generally accomplished by providing a live-stream preview from the camera on the mobile device display: the user can then reposition the device until the barcode presents in the field of view and is decoded. This task is greatly simplified with the cmbSDK as it provides a built-in preview control that can be displayed full-screen, partial screen, and in either portrait or landscape orientation. cmbSDK also supports "passive" aimers: devices that attach to the mobile device or mobile device case that use the LED flash of the device as a light source to project an aiming/targeting pattern. The advantage to these types of aimers is that an on-screen preview is no longer required (since the mobile device can now project an aimer pattern similar to a purpose-built scanner). One limitation of passive aimers, though, is that since the mobile device flash is being used for the aimer, using the LED flash for general scanning illumination is not available.

In addition to "passive" aimers, cmbSDK supports "active" aimer that is MX-100 Barcode Reader. This device is attach to the mobile device with a mobile device case, and have built in LEDs for illumination and aiming (project an green dot to help in reading the barcode).

Mobile Device Orientation

Mobile devices support developing applications for either portrait orientation, landscape orientation, or auto-rotation between the two. The cmbSDK fully supports all three options for both the presentation of the barcode preview as well as the scan direction. As mentioned previously, most barcodes can be scanned by a mobile device regardless of the orientation of the application and/or mobile device.

In some circumstances, though, using landscape orientation may be advantageous or even necessary. Mobile cameras have a higher resolution along the "height" of the image in portrait mode. For example, a common resolution used is 1280x720. When scanning barcodes in portrait mode, this means that 720 pixels of data are available for barcode decoding along the horizontal axis. If scanning a particularly long or

dense barcode (e.g. a large PDF417), using the landscape orientation provides 1280 pixels on the horizontal scan line. Orientation makes little to no difference when scanning "square" barcodes like QR, Data Matrix, and MaxiCode.

Mobile Device Performance

Today's smartphones and tablets have significant computing power. With multi-core CPUs and even dedicated image processors, they provide an ideal platform for cost-effective and efficient barcode decoding. As powerful as these devices are, developers are still advised to consider optimizing their barcode scanning applications. While the SDK has been optimized specifically for mobile environments, image analysis and barcode decoding is still a CPU intensive activity: and since these processes must share the mobile device CPU with the operating system, services, and other applications, developers should limit their applications to only using the features of the SDK that satisfy their needs.

Application optimizations include the following:

- Only enable decoding for the barcode types the application needs to scan.The cmbSDK supports the decoding of almost 40 different barcode types and subtypes, and while you can enable all of these, it can negatively impact performance as well as introduce unwanted side effects:
- The more symbologies enabled, the slower the performance. This can lead to sluggish decoding and the degradation of the overall performance of the mobile device, leaving the user with an inaccurate impression of the SDK's capabilitie.
- False reads are possible. This is particularly possible when some of the weaker symbologies, like Code 25, are enabled without proper consideration and configuration of other, more advanced features like minimum code length and barcode location. These features help mitigate false reads with the weak symbologies, but at a cost of degraded performance (and again, are not intended to all be turned on and used at the same time).
- Using an optimal camera resolution. By default, the cmbSDK uses HD images (typically 1280x720) for barcode decoding. This resolution is sufficient for all but the very smallest or dense of barcodes. As the application developer, you can use a higher resolution (full HD), but keep in mind that these images are significantly larger, so they will require more time to analyze and decode.
- Using an appropriate decoder effort level. The SDK has a user-configurable effort-level that control show aggressively the SDK performs image analysis. Like most other settings, the SDK uses a default value (level 2) that is sufficient for almost all barcodes. Using a higher level can result in better decoding of poorer quality barcodes, but at the price of slower performance.

For these reasons, when the cmbSDK is initialized for use with the built-in camera of the mobile device, no barcode symbologies are enabled by default: the application must explicitly enable the symbologies it needs. As most barcode scanning applications only truly need to scan a handful of symbologies, this behavior steers the developer to using the SDK in an efficient manner.

Enabling symbologies is a very simple process, which is explained later in this document.

cmbSDK for iOS

Getting Started

Perform the following steps to install the iOS cmbSDK:

1. Download the latest XCode for iOS Development.
2. Download the Cognex Mobile Barcode SDK for iOS.

Using the SDK in XCode

Perform the following steps to 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
* libDataManSDK.a
```

You may need to add CocoaAsyncSocket.framework (located in dependencies of cmbSDK zip file) to the "Embedded Binaries" section of the General tab in Xcode in case you want to use device discovery.

3. Go to your project's **Info.plist** file and add the Privacy - Camera Usage Description or NSCameraUsageDescription. This is required by iOS and is used to display a message explaining the usage of the user's device camera by your application.

Creating a Swift Bridging Header

If you are writing your app in Swift, you will need a Bridging Header to be able to use the cmbSDK. This is required only if your app is written in Swift, and can be done in few easy steps:

1. Create the header by selecting File -> New File -> Header File
2. Save the header file, you can name it something like: **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**". Of course, replace YourApp with your app name, and YourApp-Bridging-Header.h with your bridging headers name.
4. Open your bridging header and import the headers that you would like to use from the cmbSDK. For example, for basic functionality you will need these:

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

And you're set! You can move on to writing your app using the cmbSDK functionalities.

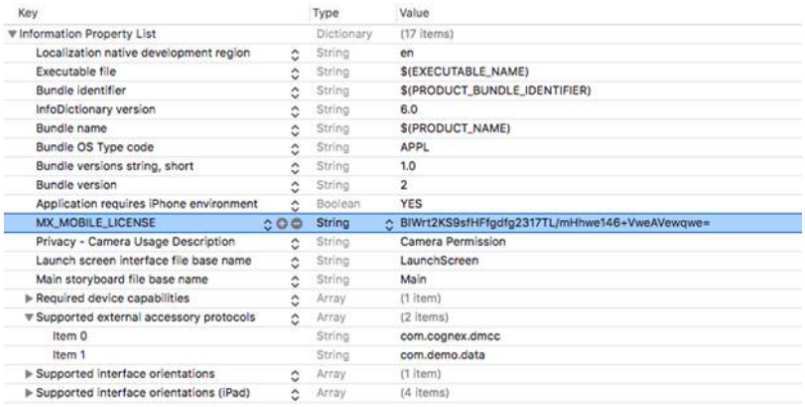
Licensing the SDK

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.

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



Key	Type	Value
▼ Information Property List	Dictionary	(17 items)
Localization native development region	String	en
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle version	String	2
Application requires iPhone environment	Boolean	YES
MX_MOBILE_LICENSE	String	BIWrt2KS9sfHFfgdfg2317TL/mHhwe146+VweAVewqwe=
Privacy - Camera Usage Description	String	Camera Permission
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
▶ Required device capabilities	Array	(1 item)
▼ Supported external accessory protocols	Array	(2 items)
Item 0	String	com.cognex.dmcc
Item 1	String	com.demo.data
▶ Supported interface orientations	Array	(1 item)
▶ Supported interface orientations (iPad)	Array	(4 items)

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 provided sample application has been written to explicitly demonstrate this capability.

Initializing a Reader Device

The cmbSDK provides two different reader class initializers: one for scanning using an MX mobile terminal (like the MX- 1000 or MX-1502) and another for scanning using the built-in camera of the mobile device (iPhones, iPads, etc).

Using the MX-1xxx Reader

Initializing the Reader Device for use with an MX mobile terminal like the MX-1000 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 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. The cmbSDK supports several configurations to provide the maximum flexibility. This includes support of optional, external aimers/illumination, as well as the ability to customize the appearance of the live-stream preview.

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**: This initializes the reader to use a live-stream preview (on the mobile device screen) so the user can position the barcode within the camera's field of view for detection and decoding. Use this mode when the mobile device does not have an aiming accessory.
- **kCDMCameraModePassiveAimer**: This initializes the reader to use a passive aimer, which is an accessory that is attached to the mobile device or a mobile device case that uses the built-in LED flash of the mobile device as a light source for projecting an aiming pattern. In this mode, no live-stream preview is presented on the device screen, since an aiming pattern will be projected.
- **kCDMCameraModeActiveAimer**: This initializes the reader to use an active aimer, such as the MX-100, which is an accessory that is attached to the mobile device or a mobile device case. The active aimer has a built in LEDs for projecting an aiming pattern, and for illumination, and are powered by a built in battery. In this mode, no live-stream preview is presented on the device screen, since an aiming pattern will be projected.
- **kCDMCameraModeFrontCamera**: This initializes the reader to use the front facing camera of the mobile device, if available (not all mobile devices have a front camera). This is an unusual, but possible configuration. Most front-facing cameras do not have auto focus and illumination, and provide significantly lower resolution images. This option should be used with care. In this mode, illumination is not available.

All of the above modes provide the following default settings for the reader:

- The rear camera is used.
- The zoom feature is available and a button to control it is visible on the live-stream preview (if displayed).
- 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 is available and a button to control it is visible on the live-stream preview.
  - If commands are sent to the reader for aimer control, they will be ignored.

- **kCDMCameraModePassiveAimer**(PassiveAimer)
  - The live-stream preview will not be displayed when the startScanning() method is called.
  - Illumination is not available and the live-stream preview will not have an illumination button.
  - If commands are sent to the reader for illumination control, they will be ignored, since it is assumed in this mode that the built-in LED of the mobile device is being used for the aimer.

- **kCDMCameraModeActiveAimer**(MX-100)
  - The live-stream preview will not be displayed when the startScanning() method is called.
  - Illumination is available and if a preview option for camera preview is used, the preview will have an illumination button.
  - If commands are sent to the reader for illumination or aimer control, they will be accepted.

- **kCDMCameraModeFrontCamera**(FrontCamera)
  - The live-stream preview is displayed when the startScanning() method is called.
  - The front camera is used.
  - Illumination is not available, and the live-stream preview will not have an illumination button. o If commands are sent to the reader for aimer or illumination control, they will be 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**: Use this option to accept all defaults set by the CameraMode.
- **kCDMPreviewOptionNoZoomBtn**: This option hides the zoom button on the live-stream preview, preventing a user from adjusting the zoom of the mobile device camera.
- **kCDMPreviewOptionNoIllumBtn**: This hides the illumination button on the live-stream preview, preventing a user from toggling the illumination.
- **kCDMPreviewOptionHwTrigger**: This enables a simulated hardware trigger (the volume down button)for starting scanning on the mobile device. This button only starts scanning when pressed. It does not need to be held like a more traditional purpose-built scanner's trigger. Pressing the button a second time does not stop the scanning process.
- **kCDMPreviewOptionPaused**: If using a live-stream preview, when this option is set, the preview will be displayed when the startScanning() method is called, but the reader will not start decoding (i.e. looking for barcodes) until the user presses the on-screen scanning button to actually start the scanning process.
- **kCDMPreviewOptionAlwaysShow**: This forces alive-stream preview to be displayed, even if an aiming mode has been selected (e.g. *CameraMode == kCDMCameraModePassiveAimer*)
- **kCDMPreviewOptionPessimisticCaching**: Used 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**: This will use the device camera in higher resolution. The default resolution is 1280x720. With this option is enabled, the resolution will be 1920x1080 on devices that support it, and the default one on devices that do not. This can help with scanning small barcodes, but will increase the decoding time since there is a lot more data to process in each frame.
- **kCDMPreviewOptionHighFrameRate**: This will use the device camera in 60 FPS instead of the default 30, and provide a much smoother camera preview.

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:CDMPrev
readerDevice.delegate = self
```

```
CMBReaderDevice *readerDevice = [CMBReaderDevice readerOfDeviceCameraWithCameraMode:kCDMCameraModeNoAimer previewOptions:kC
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:[CDMPre
readerDevice.delegate = self
```

```
CMBReaderDevice *readerDevice = [CMBReaderDevice readerOfDeviceCameraWithCameraMode:kCDMCameraModeNoAimer previewOptions:(k
readerDevice.delegate = self;
```

Connecting to the Device

After initializing the Reader Device and setting a delegate to handle responses from the reader, you are ready to 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
```

```
        }
    }];
}
```

If everything was done correctly, *connectionStateDidChangeOfReader* in the delegate will be 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 Reader Device, and can begin using the Cognex Mobile Barcode SDK.

Configuring the Reader Device

After connecting to the scanning device, you may want (or need) to change some of its settings. The cmbSDK provides a set of high-level, device independent APIs for setting and retrieving the current configuration of the device.

Like in the case of initializing the Reader Device, there are some differences between using an MX reader and the camera reader for scanning. These differences are detailed in the following sections.

MX Mobile Terminals

The MX family of mobile terminals provides sophisticated device configuration and management, including saved configurations on the device itself. In general, these devices come from Cognex preconfigured for an exceptional out-of- the-box experience with most symbologies and features ready to use.

When custom reconfiguration is desired, this is typically done using either the DataMan Setup Tool, or the Cognex Quick Setup as these tools can be used to distribute saved configurations easily to multiple devices, thereby greatly simplifying configuration management.

However, it is still possible (and sometimes desirable) for the mobile application itself to configure the MX device:

- You can have multiple scanning applications, each of which requires a different set of device settings.
- You may simply want to guarantee that the certain options are in a “known” state and not rely on the fact that the device has been pre-configured correctly.

Built-in Camera

Much like an MX mobile terminal, the cmbSDK employs a default set of options for barcode reading with the built-in camera of the mobile device, providing a good out-of-box experience. However, there are two important differences to keep in mind:

- The cmbSDK does not implement saved configurations for the camera reader. This means that every time an application that uses the camera reader starts, it starts with its defaults.
- The cmbSDK does not enable any symbologies by default: you as the application programmer must enable all barcode symbologies your application needs to scan. By requiring the application program to explicitly enable only the symbologies it needs, the most optimal scanning performance can be achieved. This concept was more thoroughly discussed in the Overview section.

MX-100

MX-100 is a device-case attachment that provides additional functionalities to the built-in camera, like aiming and better illumination control. Being a hybrid of an MX device and a built in scanner, it has it's own settings for aimer intensity, illumination intensity and aimer modulation stored on the device itself, and the rest of the settings, like symbologies, stored in the cmbSDK.

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

- By default, MX-100 does not require a license to work with the device camera, but one can be generated for tracking purposes, free of charge.
- MX-100 comes pre-configured for a better out-of-the-box experience, and the cmbSDK has the following symbologies enabled by default:
    - Code 39
    - Code 128
    - Databar
    - PDF417
    - QR
    - UPC/EAN

- 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 itself. Different settings are stored in different cache:

  - *Persistent cache*: These are settings/values that do not change very often (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*: These are settings/values that may change while an application is using an MX-100 (though not likely), but should be read from the MX-100 at least on SDK load/initial connection to the MX-100. These items are: Aimer intensity, Aimer modulation, Aimer timeout, Illumination intensity, Illumination state. When the SDK initially connects to an MX-100, these values are read from the MX-100.
    By default, the session cache will be maintained optimistically for the best performance. By this, we mean that the SDK will assume that another application is not changing the settings of the aimer (SDK only needs to read the aimer's settings one time, when the initial connection is established). If this happens, the cache may become out of sync with the aimer (e.g. another app on the same device changes a setting on the aimer). cmbSDK gives the possibility to handle the session cache *pessimistically* which means that above written aimer's configuration is loaded again each time when application is resumed. This behavior is accomplished by adding an additional option flag to the camera connector: **kCDMPreviewOptionPessimisticCaching**.

Enabling Symbologies

Individual symbologies can be enabled using the following method of the Reader Device 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
    }
})
```

```
[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
    }
}];
```

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
    }
})
```

```
[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
    }
}];
```

Illumination Control

If your reader device is equipped with illumination lights (e.g. LEDs), you can control whether they are ON or OFF when scanning starts using the following method of your Reader Device 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 or cable unplugged, or c
    }
}
```

```
[readerDevice setLightsON:YES completion:^(NSError *error) {
    if (error) {
        // Failed to enable illumination, Possible causes are: reader disconnected, out of battery or cable unplugged, or c
    }
}];
```

Keep in mind that not all devices and device modes supported by the cmbSDK allow for 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% on iOS)
- level 2 zoom (default 400% on iOS)

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.

If the scanning is finished, the values is reset to normal behavior (0).

Example

- Swift
- Objective-C

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

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

Resetting the Configuration

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 (not the factory defaults), 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 or cable unplugged
    }
}
```

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

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. Change the scan direction to omnidirectional:

- Swift
- Objective-C

```
self.readerDevice.dataManSystem()?.sendCommand("SET DECODER.1D-SYMBOLORIENTATION 0", withCallback: { (response:CDMResponse?
    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:^(CDMResponse *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?) i
    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
    }
}];
```

Camera Overlay Customization

When using the Mobile Camera, 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 in many ways. The cmbSDK camera overlay is built from buttons for zoom, flash and closing the scanner (in full screen), a progress bar indicating the scan timeout, and lines on the corners of the camera preview.

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 readerDevice:

- Swift
- Objective-C

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

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

The LEGACY overlay has limited customizability, so it is preferred to use the CMB overlay.

When using the CMB overlay, you can copy the layout files found in the Resources/layout directory into your project and modify them as you like. The files are: **CMBScannerPartialView.xib** used when the scanner is started inside a container (partial view), and **CMBScannerView.xib** when the scanner is started in full screen.
After copying the layout that you need, or both layouts, you can modify them, for example by changing the sizes, positions or color of the views, removing views and even 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, so while you can change everything about those views, make sure the *Tag* attribute remains unchanged, otherwise the cmbSDK will not be able to recognize those views and continue to function as if those views were removed.

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];
```

Scanning Barcodes

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

- If using an MXreader, the user can now 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 decodeda barcode.
- The user released the trigger or pressed the stop button on the live-stream preview screen.
- The camera reader timed out with out finding a barcode.
- The application program itself calls the *stopScanning()* method.

When a barcode is decoded successfully (the first case), you will receive a CMBReadResults array in your Reader Device'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;
```

The following is an example 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.

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?,
    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 qualifie
    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.

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 Disconnects

1. Disconnects:

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 you return to your application from inactive state, the reader device remains initialized, but not connected. This means there is no need for reinitializing the SDK, but you will need to re-connect.

Some iOS versions will send a "Availability" notification when resuming the application that the External Accessory is available. You can use this in the CMBReaderDeviceDelegate's method: (void)availabilityDidChangeOfReader:(CMBReaderDevice *)reader. In it, when the reader becomes available, you can connect to it.

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 will have to handle this manually. For this, you will have to add an observer for "UIApplicationDidBecomeActiveNotification" and do some checks before connecting, so you don't connect while the reader is already in "connecting" or "connected" state. For example:

- Swift
- Objective-C

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Reconnect when app resumes
    NotificationCenter.default.addObserver(self, selector: #selector(self.appBecameActive), name:NSNotification.Name.UIAppl
```

```
}

// handle app resume
func appBecameActive() {
    if readerDevice != nil
        && readerDevice.availability == CMBReaderAvailibilityAvailable
        && readerDevice.connectionState != CMBConnectionStateConnecting && readerDevice.connectionState != CMBConnectionSta
        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 object:nil];
}

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

Appendix A - DMCC for the Camera Reader - iOS

Appendix A - DMCC for the Camera Reader - iOS

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

Note: Many of these commands are also supported by the MX mobile terminals. Commands that are unique to the camera reader or MX-100 are indicated as such with an X in the last column.

GET/SET	COMMAND	PARAMETER(S)	DESCRIPTION	CAMERAREADER
GET/SET	BATTERY.CHARGE		Returns the current battery level of the device as a percentage.	
	BEEP		Plays the audible beep (tone).	

GET/SET	COMMAND	PARAMETER(S)	DESCRIPTION	CAMERAREADER
GET/SET	BEEP.GOOD	[0-3] [0-2]	Sets the number of beeps (0-3) and the beep tone/pitch (0-2, for low, medium, high). For the built-in camera, only a single beep with no pitch control is supported. Thus, 0 1 turns the beep off, 1 1 turns the beep on.	
GET/SET	CAMERA.ZOOM	0-2	The possible values for the SET command are: 0 - normal (un-zoomed), 1 - zoom at level 1, 2 - zoom at level 2. This zoom level is used during scanning. When scanning ends it reset to 0.	X
GET/SET	CAMERA.ZOOM-PERCENT	[100-MAX] [100-MAX]	Sets/Returns **level 1** zoom (default 200% on iOS, 150% on Android), and **level 2** zoom (default 400% on iOS, 300% on Android). Note: The camera needs to be started at least once from sdk to have a proper value for max capable zoom (MAX)	X
GET/SET	CODABAR.CODESIZE	ON min max OFF min max	Accepts any length Codabar. Sets min/max length of accepted Codabar.	X X
GET/SET	C11.CHKCHAR	ON | OFF	Turns Code 11 check digit on/off.	X
GET/SET	C11.CHKCHAR-OPTION	1 2	Requires single checksum. Requires double checksum.	X X
GET/SET	C11.CODESIZE	ON min max OFF min max	Accepts any length Code 11. Sets min/max length of accepted Code 11.	X X
GET/SET	C25.CODESIZE	ON min max OFF min max	Accepts any length Code 25. Sets min/max length of accepted Code 25.	X X
GET/SET	C39.ASCII	ON | OFF	Turns Code 39 extended ASCII on/off.	
GET/SET	C39.CODESIZE	ON min max OFF min max	Accepts any length Code 39. Sets min/max length of accepted Code 39.	
GET/SET	C39.CHKCHAR	ON | OFF	Turns Code 39 check digit on/off	
GET/SET	C93.ASCII	ON | OFF	Turns Code 93 extended ASCII on/off	X

GET/SET	COMMAND	PARAMETER(S)	DESCRIPTION	CAMERAREADER
GET/SET	C93.CODESIZE	ON min max OFF min max	Accepts any length Code 93. Sets min/max length of accepted Code 93.	
	CONFIG.DEFAULT		Resets most of the camera API settings to default, except those noted as not resetting (see Appendix B). To reset all settings, use DEVICE.DEFAULT.	
GET/SET	DATA.RESULT-TYPE	0 1 2 4 8	Specifies results to be returned (sum of multiple values): None Text string result (default) XML results XML stats Scan image (see IMAGE.* commands)	
GET/SET	DATABAR.EXPANDED	ON | OFF	Turns the DataBar Expanded symbology on/off.	
GET/SET	DATABAR.LIMITED	ON | OFF	Turns the DataBar Limited symbology on/off.	
GET/SET	DATABAR.RSS14	ON | OFF	Turns the DataBar RSS14 symbology on/off.	X
GET/SET	DATABAR.RSS14STACK	ON | OFF	Turns the DataBar RSS14 Stacked symbology on/off.	X
GET/SET	DECODER.1D-SYMBOLORIENTATION	0 1 2 3	Use omnidirectional scan orientation. Use horizontal and vertical scan orientation. Use vertical scan orientation. Use horizontal scan orientation.	
GET/SET	DECODER.EFFORT	1-5	Sets the effort level for image analysis/decoding. The default is 2. Do not use 4-5 for online scanning.	X
GET/SET	DECODER.MAX-SCAN-TIMEOUT	1-120	Sets the timeout for the live-stream preview. When the timeout is reached, decoding is paused; the live-stream preview will remain on-screen.	X
GET	DECODER.MAX-THREADS		Returns the max number of CPU threads supported by the device.	X
GET/SET	DECODER.THREADS-USED	[0-MAX]	Specify the max number of CPU threads that the scanner can use during the scanning process.	X

GET/SET	COMMAND	PARAMETER(S)	DESCRIPTION	CAMEREADER
	DEVICE.DEFAULT		Resets the camera API settings to default (see Appendix B).	
GET	DEVICE.FIRMWARE-VER		Gets the device firmware version.	
GET	DEVICE.ID		Returns device ID assigned by Cognex to the scanning device. For a built-in camera, SDK returns 53. For MX-100 Barcode Scanner, SDK returns 56.	
GET/SET	DEVICE.NAME		Returns the name assigned to the device. By default, this is “MX-“ plus the last 6 digits of DEVICE.SERIAL-NUMBER.	
GET	DEVICE.SERIAL-NUMBER		Returns the serial number of the device. For a built-in camera, the SDK assigns a pseudo-random number.	
GET	DEVICE.TYPE		Returns the device name assigned by Cognex to the scanning device. For a built-in camera, SDK returns “MX-Mobile”. If MX-100 is available, SDK returns "MX-100".	
GET/SET	FOCUS.FOCUSTIME	0-10	Sets the camera’s auto-focus period (how often the camera should attempt to refocus). The default is 3, but it is 1 for MX-100.	
GET/SET	I2O5.CHKCHAR	ON | OFF	Turns Interleaved 2 of 5 check digit on/off.	
GET/SET	I205.CODESIZE	ON min max OFF min max	Accepts any length Interleaved 2 of 5. Sets min/max length of accepted Interleaved 2 of 5.	X X
GET/SET	IMAGE.FORMAT	0 1 2	Scanner returns image result in bitmap format. Scanner returns image result in JPEG format. Scanner returns image result in PNG format.	
GET/SET	IMAGE.QUALITY	10, 15, 20, ...90	Specifies JPEG image quality.	
GET/SET	IMAGE.SIZE	0 1 2 3	Scanner returns full size image. Scanner returns 1⁄4 size image. Scanner returns 1/16 size image. Scanner returns 1/62 size image.	

GET/SET	COMMAND	PARAMETER(S)	DESCRIPTION	CAMEREADER
GET/SET	LIGHT.AIMER	0-1	Disables/enables the aimer (when the scanner starts).	
SET	LIGHT.AIMER-CONFIG	[32-100] [0-15] [32-100]	Sets MX-100's configuration; parameters are aimer intensity, aimer modulation, illumination intensity	X
GET	LIGHT.AIMER-CONFIG	0 | 1	Get's all of the MX-100 configuration parameters (see above) at one time. Using option 0 reads the settings from the cache; using option 1 always reads from the device.	X
GET/SET	LIGHT.AIMER-INTENSITY	32-100	Sets/gets the aimer LED's intensity (as a percentage)	X
GET/SET	LIGHT.AIMER-MODULATION	0-15	Sets/gets the aimer LED's modulation (blink rate); parameter is milliseconds	X
GET/SET	LIGHT.AIMER-TIMEOUT	0-600	Timeout in seconds for an aimer. This value is always overridden by DECODER.MAX-SCAN-TIMEOUT.	
GET/SET	LIGHT.INTERNAL-ENABLE	ON | OFF	Enables/disables illumination (when the scanner starts).	
GET/SET	MSI.CHKCHAR	ON | OFF	Turns MSI Plessey check digit on/off.	
GET/SET	MSI.CHKCHAR-OPTION	0 1 2 3 4 5	Use mod 10 checksum Use mod 10 mod 10 checksum Use mod 11 checksum (IBM algorithm) Use mod 11 mod 10 checksum (IBM algorithm) Use mod 11 checksum (NCR algorithm) Use mod 11 mod 10 checksum (NCR algorithm)	X X
GET/SET	MSI.CODESIZE	ON min max OFF min max	Accepts any length MSI Plessey. Sets min/max length of accepted MSI Plessey.	X X
GET/SET	SYMBOL.AZTECCODE	ON | OFF	Turns the Aztec Code symbology on/off.	
GET/SET	SYMBOL.CODABAR	ON | OFF	Turns the Codabar symbology on/off.	
GET/SET	SYMBOL.C11	ON | OFF	Turns the Code 11 symbology on/off.	X

GET/SET	COMMAND	PARAMETER(S)	DESCRIPTION	CAMEREADER
GET/SET	SYMBOL.C128	ON | OFF	Turns the Code 128 symbology on/off.	
GET/SET	SYMBOL.C25	ON | OFF	Turns the Code 25 symbology on/off (standard).	
GET/SET	SYMBOL.C39	ON | OFF	Turns the Code 39 symbology on/off.	
GET/SET	SYMBOL.C93	ON | OFF	Turns the Code 93 symbology on/off.	
GET/SET	SYMBOL.COOP	ON | OFF	Turns the COOP symbology (Code 25 variant) on/off.	X
GET/SET	SYMBOL.DATAMATRIX	ON | OFF	Turns the Data Matrix symbology on/off.	
GET/SET	SYMBOL.DATABAR	ON | OFF	Turns the DataBar Expanded and Limited symbologies on/off.	
GET/SET	SYMBOL.DOTCODE	ON | OFF	Turns the DotCode symbology on/off.	
GET/SET	SYMBOL.IATA	ON | OFF	Turns the IATA symbology (Code 25 variant) on/off.	X
GET/SET	SYMBOL.INVERTED	ON | OFF	Turns the Inverted symbology (Code 25 variant) on/off.	X
GET/SET	SYMBOL.ITF14	ON | OFF	Turns the ITF-14 symbology (Code 25 variant) on/off.	X
GET/SET	SYMBOL.UPC-EAN	ON | OFF	Turns the UPC-A, UPC-E, EAN-8, and EAN-13 symbologies on/off.	
GET/SET	SYMBOL.MATRIX	ON | OFF	Turns the Matrix symbology (Code 25 variant) on/off.	X
GET/SET	SYMBOL.MAXICODE	ON | OFF	Turns the MaxiCode symbology on/off.	X
GET/SET	SYMBOL.MSI	ON | OFF	Turns the MSI Plessey symbology on/off.	
GET/SET	SYMBOL.PDF417	ON | OFF	Turns the PDF417 symbology on/off.	
GET/SET	SYMBOL.PLANET	ON | OFF	Turns the PLANET symbology on/off.	

GET/SET	COMMAND	PARAMETER(S)	DESCRIPTION	CAMERAREADER
GET/SET	SYMBOL.POSTNET	ON | OFF	Turns the POSTNET symbology on/off.	
GET/SET	SYMBOL.4STATE-IMB	ON | OFF	Turns the Intelligent Mail Barcode symbology on/off.	
GET/SET	SYMBOL.4STATE-RMC	ON | OFF	Turns the Royal Mail Code symbology on/off.	
GET/SET	SYMBOL.QR	ON | OFF	Turns the QR and MicroQR symbologies on/off.	
GET/SET	TRIGGER.TYPE	0 1 2 3 4 5	Not supported Not supported Manual (default) Not supported Not supported Continuous	
GET/SET	UPC-EAN.EAN13	ON | OFF	Turns the EAN-13 symbology on/off.	X
GET/SET	UPC-EAN.EAN8	ON | OFF	Turns the EAN-8 symbology on/off.	X
GET/SET	UPC-EAN.UPC-A	ON | OFF	Turns the UPC-A symbology on/off.	X
GET/SET	UPC-EAN.UPC-E	ON | OFF	Turns the UPC-E symbology on/off.	X
GET/SET	UPC-EAN.UPCE1	ON | OFF	Turns the UPC-E1 symbology on/off.	
GET/SET	UPCE-AN.SUPPLEMENT	0 1-4	Turns off UPC supplemental codes. Turns on UPC supplemental codes.	
GET/SET	VIBRATION.GOOD	ON | OFF	Sets/gets whether to vibrate when a code is read (default is ON)	

Appendix B - Camera Reader Defaults - iOS

Appendix B - Camera Reader Defaults - iOS

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.DEFAULT 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 UPCE- AN.UPCE1	ON OFF OFF OFF ON ON ON ON OFF	

SETTING	DEFAULT VALUE	DEVICE RESET ONLY?
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	

Precautions - iOS

Precautions - iOS

Observe these precautions when installing the Cognex product, to reduce the risk of injury or equipment damage:

- To reduce the risk of damage or malfunction due to over-voltage, line noise, electrostatic discharge (ESD), power surges, or other irregularities in the power supply, route all cables and wires away from high-voltage power sources.
- Changes or modifications not expressly approved by the party responsible for regulatory compliance could void the user’s authority to operate the equipment.
- Cable shielding can be degraded or cables can be damaged or wear out more quickly if a service loop or bend radius is tighter than 10X the cable diameter. The bend radius must begin at least six inches from the connector.
- This device should be used in accordance with the instructions in this manual.
- All specifications are for reference purpose only and may be changed without notice.