Xamarin (v2.4.x)

Introduction

Xamarin is unique by offering a single language – C#, class library, and runtime that works across all three mobile platforms of iOS, Android, and Windows Phone (Windows Phone's native language is already C#), while still compiling native (non-interpreted) applications that are performant enough even for demanding games.

Each of these platforms has a different feature set and each varies in its ability to write native applications – that is, applications that compile down to native code and that interop fluently with the underlying Java subsystem. For example, some platforms only allow apps to be built in HTML and JavaScript, whereas some are very low-level and only allow C/C++ code. Some platforms don't even utilize the native control toolkit.

Xamarin is unique in that it combines all of the power of the native platforms and adds a number of powerful features of its own, including:

1. **Complete Binding for the underlying SDKs** – Xamarin contains bindings for nearly the entire underlying platform SDKs in both iOS and Android. Additionally, these bindings are strongly-typed, which means that they're easy to navigate and use, and provide robust compile-time type checking and during development. This leads to fewer runtime errors and higher quality applications.
2. **Objective-C, Java, C, and C++ Interop** – Xamarin provides facilities for directly invoking Objective-C, Java, C, and C++ libraries, giving you the power to use a wide array of 3rd party code that has already been created. This lets you take advantage of existing iOS and Android libraries written in Objective-C, Java or C/C++. Additionally, Xamarin offers binding projects that allow you to easily bind native Objective-C and Java libraries using a declarative syntax.
3. **Modern Language Constructs** – Xamarin applications are written in C#, a modern language that includes significant improvements over Objective-C and Java such as *Dynamic Language Features* , *Functional Constructs* such as *Lambdas* , *LINQ* , *Parallel Programming* features, sophisticated *Generics* , and more.
4. **Amazing Base Class Library (BCL)** – Xamarin applications use the .NET BCL, a massive collection of classes that have comprehensive and streamlined features such as powerful XML, Database, Serialization, IO, String, and Networking support, just to name a few. Additionally, existing C# code can be compiled for use in an applications, which provides access to thousands upon thousands of libraries that will let you do things that aren't already covered in the BCL.
5. **Modern Integrated Development Environment (IDE)** – Xamarin uses Xamarin Studio on Mac OS X and Visual Studio on Windows. These are both modern IDE's that include features such as code auto completion, a sophisticated Project and Solution management system, a comprehensive project template library, integrated source control, and many others.
6. **Mobile Cross Platform Support** – Xamarin offers sophisticated cross-platform support for the three major mobile platforms of iOS, Android, and Windows Phone. Applications can be written to share up to 90% of their code, and our Xamarin.Mobile library offers a unified API to access common resources across all three platforms. This can significantly reduce both development costs and time to market for mobile developers that target the three most popular mobile platforms.

How Does Xamarin Work?

Xamarin offers two commercial products: Xamarin.iOS and Xamarin.Android. They're both built on top of *Mono*, an open-source version of the .NET Framework based on the published .NET ECMA standards. Mono has been around almost as long as the .NET framework itself, and runs on nearly every imaginable platform including Linux, Unix, FreeBSD, and Mac OS X.

On iOS, Xamarin's *Ahead-of-Time* (*AOT*) Compiler compiles Xamarin.iOS applications directly to native ARM assembly code. On Android, Xamarin's compiler compiles down to *Intermediate Language* (*IL*), which is then *Just-in-Time* (*JIT*) compiled to native assembly when the application launches.

In both cases, Xamarin applications utilize a runtime that automatically handles things such as memory allocation, garbage collection, underlying platform interop, etc.

Xamarin.Forms

Xamarin.Forms is a framework that allows developers to rapidly create cross platform user interfaces. It provides it's own abstraction for the user interface that will be rendered using native controls on iOS, Android, Windows, or Windows Phone. This means that applications can share a large portion of their user interface code and still retain the native look and feel of the target platform.

Xamarin.Forms allows for rapid prototyping of applications that can evolve over time to complex applications. Because Xamarin.Forms applications are native applications, they don't have the limitations of other toolkits such as browser sandboxing, limited APIs, or poor performance. Applications written using Xamarin.Forms are able to utilize any of the API's or features of the underlying platform, such as (but not limited to) CoreMotion, PassKit, and StoreKit on iOS; NFC and Google Play Services on Android; and Tiles on Windows. In addition, it's possible to create applications that will have parts of their user interface created with Xamarin.Forms while other parts are created using the native UI toolkit.

Xamarin.Forms applications are architected in the same way as traditional cross-platform applications. The most common approach is to use Portable Libraries or Shared Projects to house the shared code, and create platform specific applications that will consume the shared code.

There are two techniques to create user interfaces in Xamarin.Forms. The first technique is to create UIs entirely with C# source code. The second technique is to use *Extensible Application Markup Language* (XAML), a declarative markup language that is used to describe user interfaces. For more information about XAML, see XAML Basics.

Instalation

To start developing Xamarin application first you need to install Visual Studio or Xamarin Studio and make sure to include all necessary Xamarin components. In our examples we will use Visual Studio to show you how to develop Xamarin application and use our SDK. Navigate to this link to read step by step how to download and install Visual Studio for Xamarin applications.

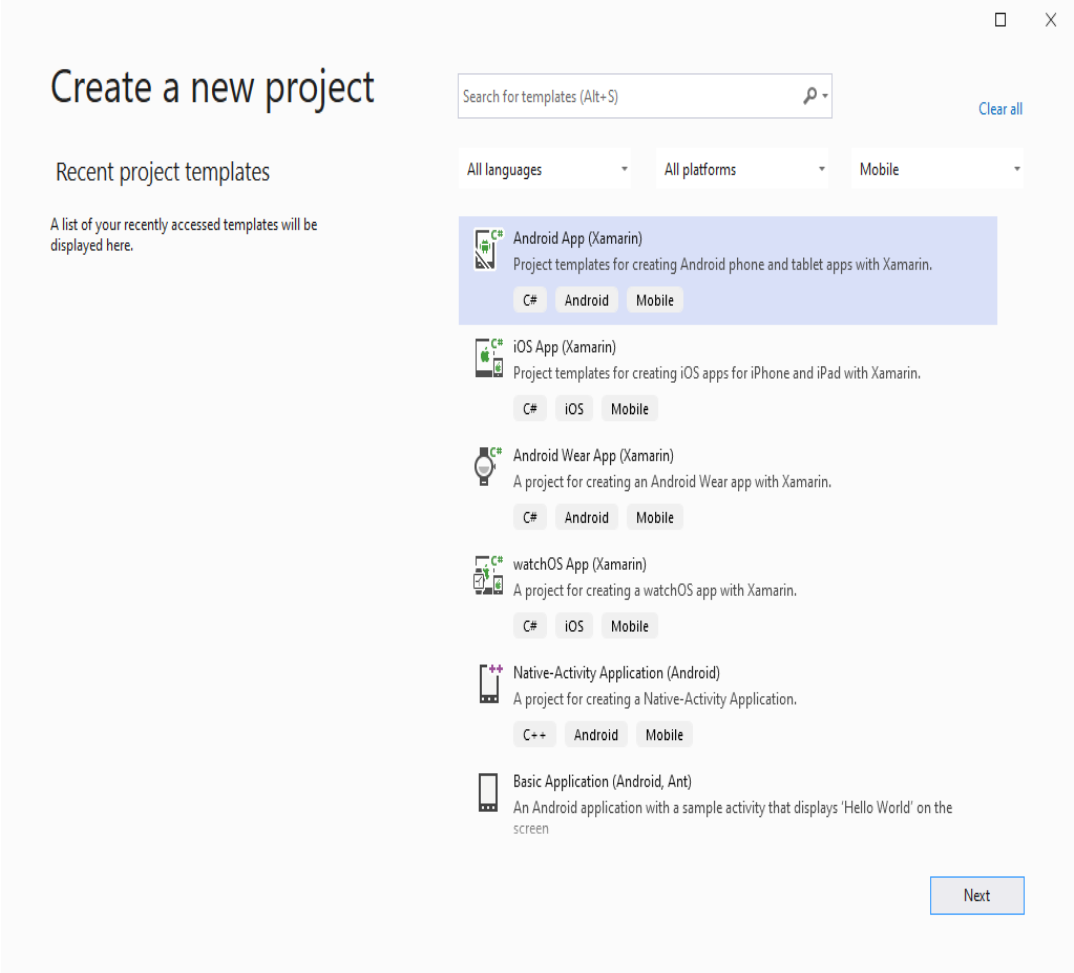
Xamarin.Android

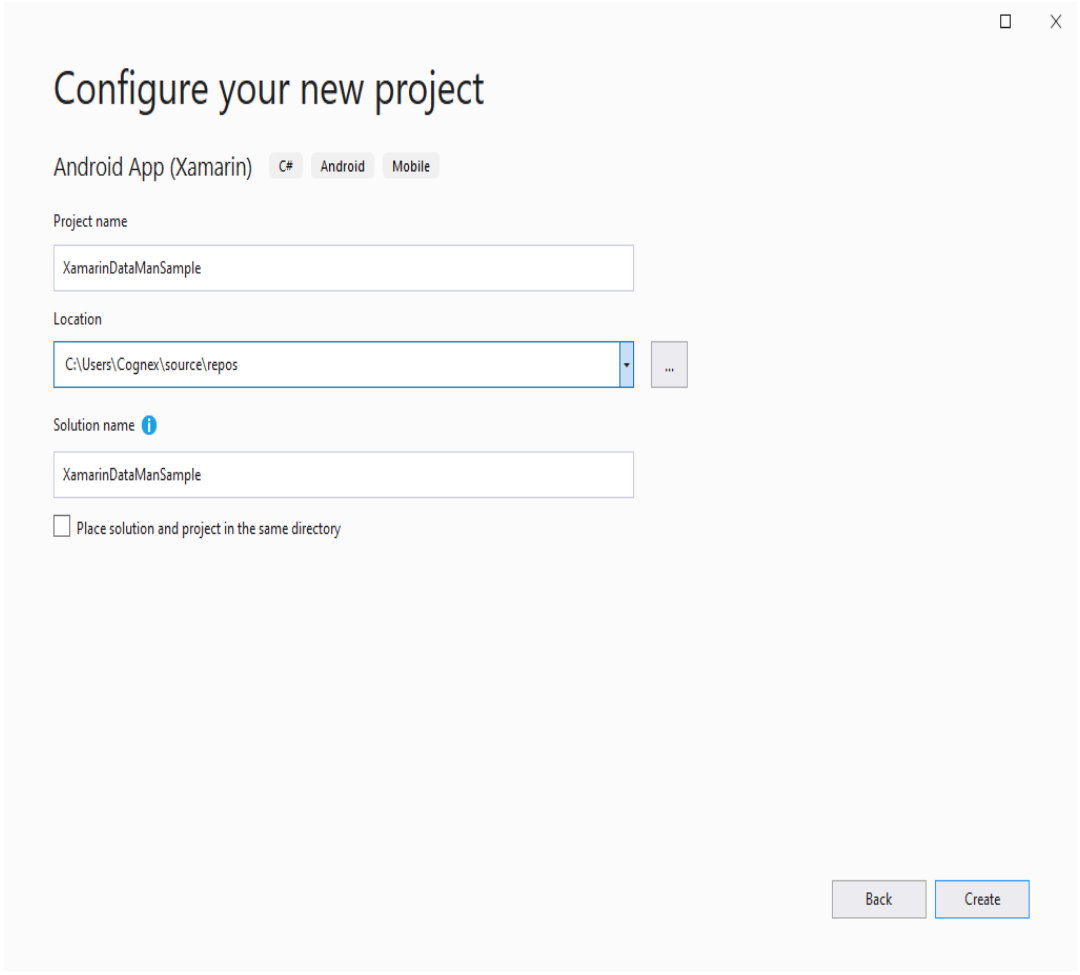
Getting Started

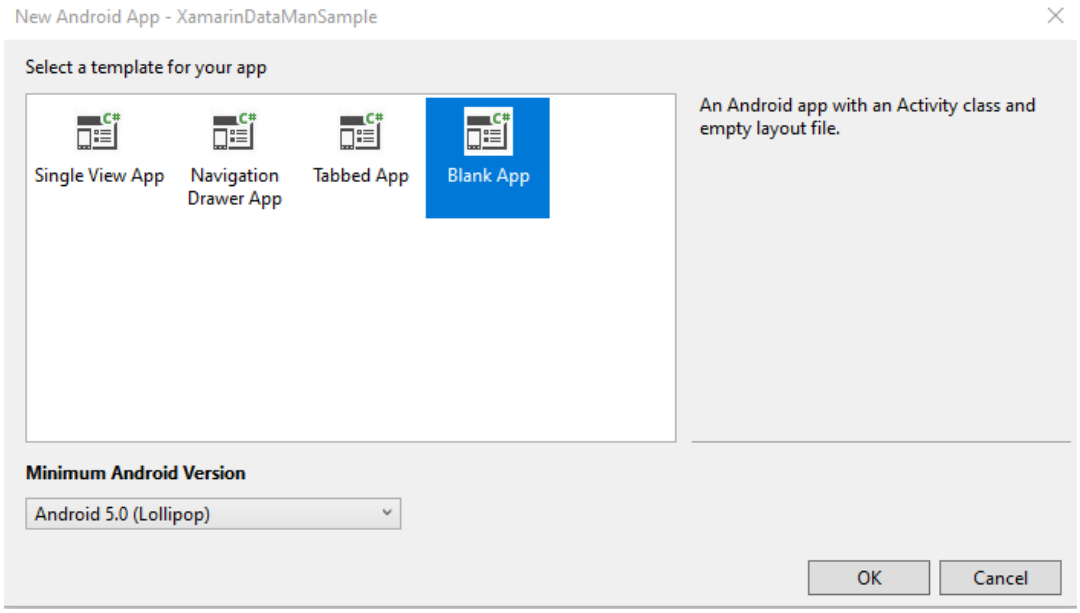
Open Visual Studio and follow these steps:

1. Go to **File -> New -> Project.**

2. Select **Android App (Xamarin),** set project name and from templates select **Blank App**



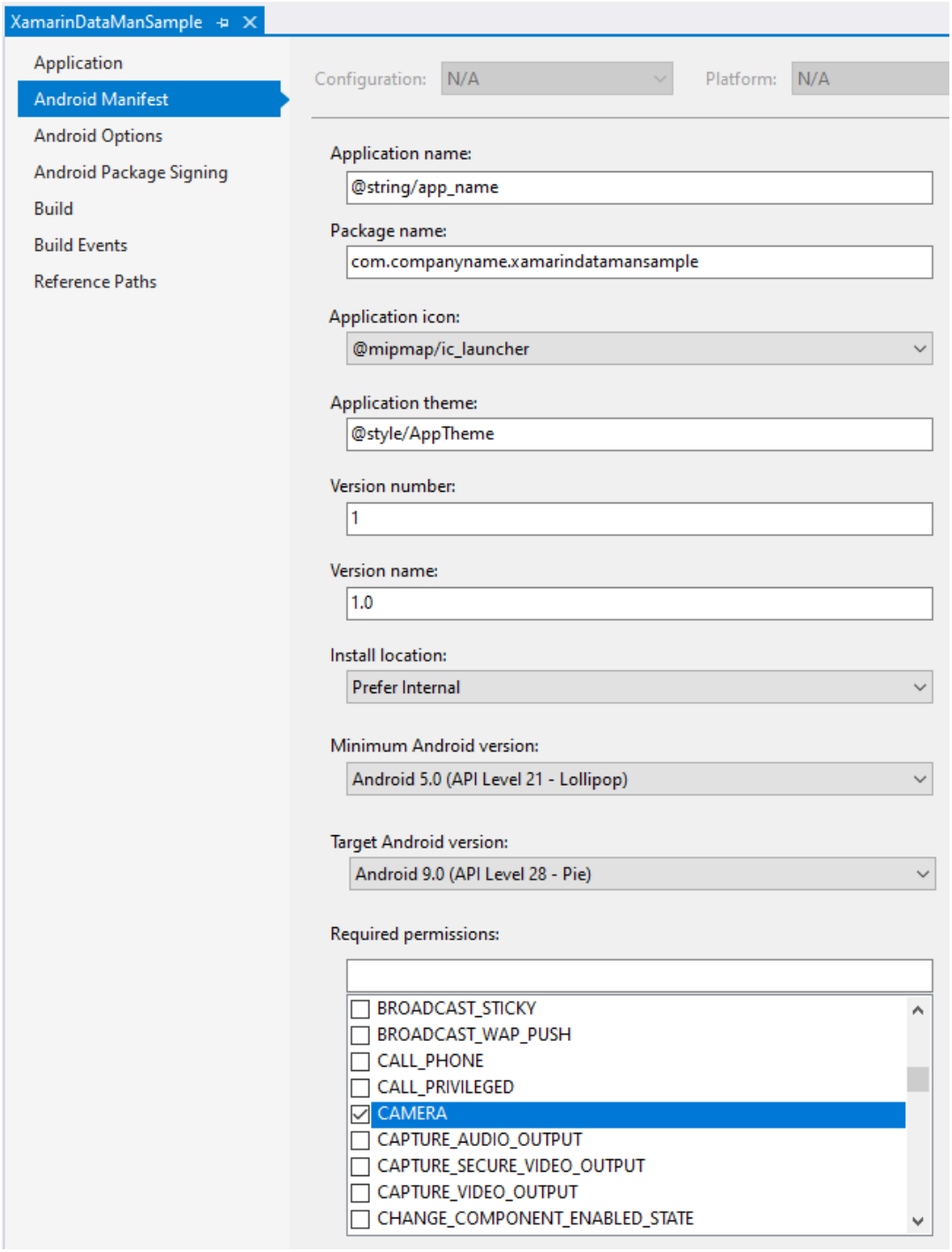




After creating a blank application, create all resources you will use (icons, images, styles, layouts, etc.). You can copy them from our sample.

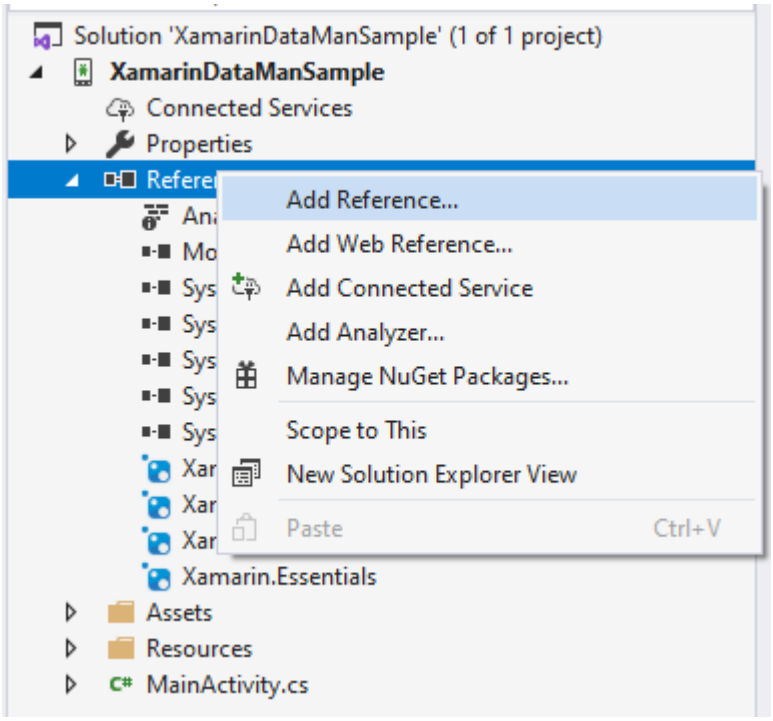
Next right click on your project file, then click Properties and go to Android Manifest section.

Setup your Manifest file (app name, app icon, minimum and maximum android versions), make sure to enable Camera permission for this application.

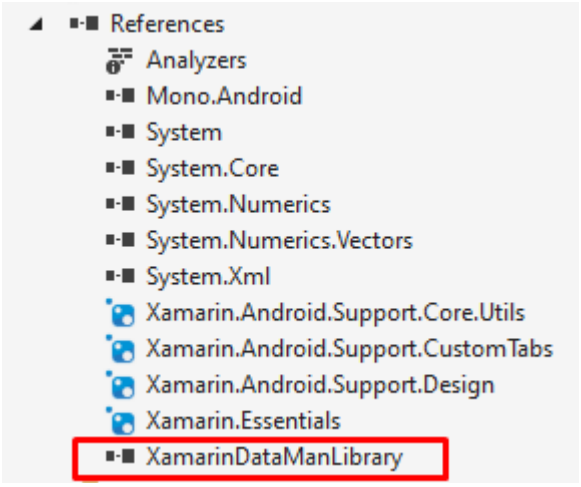


Reference cmbSDK

Now we need to reference **XamarinDataManLibrary.dll** in order to use the cmbSDK.



Browse and find **XamarinDataManLibrary.dll** file and add as reference in this project.



Licensing the SDK

To use cmbSDK for barcode scanning with a mobile device without an MX mobile terminal, you need to install a license key. If the license key is missing, asterisks will appear instead of scanned results.

Contact your Cognex Sales Representative for information on how to obtain a license key, including 30-day trial licenses.

After obtaining your license key there is two ways to add your license key in application.

The first one is to implement an activation directly from the code when you create your readerDevice:

```
...
if (param_deviceClass == DataManDeviceClass.PhoneCamera)
{
                //*****************************************************************************************
                // Create a camera reader
                //
                // NOTE:  if we're scanning using the built-in camera
```

```
                //       of the mobile phone or tablet, then the SDK requires a license key. Refer to
                //       the SDK's documentation on obtaining a license key as well as the methods for
                //       passing the key to the SDK (in this example, we're relying on an entry in
                //       AndroidManifest--there are also getPhoneCameraDevice methods where it can be passed
                //       as a parameter).
                //*************************************************************************************
                readerDevice = ReaderDevice.GetPhoneCameraDevice(this, param_cameraMode, PreviewOption.Defaults, null, "SDK
}
```

and second is to add the following line in the AndroidManifest.xml file of your application under the application tag:

```
<application>
        .....

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

Implementing SDK

1. First build your UI according to your needs, but considering the following aspects:

- You have to decide if you you want to show partial or a full screen (that is the default) camera preview. You need a **ViewGroup** container to use partial preview, for example **RelativeLayout.** No additional container is needed for full screen preview.

Our sample app (that you can find in the cmbSDK bundle) is using full screen preview. To change the sample app to use partial view, add the following **RelativeLayout** at the end inside main RelativeLayout in **activity_scanner.xml** file. Use this layout as a **ViewGroup** parameter in reader device constructor (**getPhoneCameraDevice**) when reader device is initialized.

```
<RelativeLayout
        android:id="@+id/rlPreviewContainer"
        android:layout_width="match_parent"
        android:layout_height="200dp"
        android:layout_alignParentTop="true" />
```

- To display the last scanned image, an **ImageView** container is needed.
- To display the scanned result as a text, a **TextView** is needed.

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

```
public class ScannerActivity : AppCompatActivity, IOnConnectionCompletedListener, IReaderDeviceListener, IOnSymbologyLister
        ActivityCompat.IOnRequestPermissionsResultCallback {

...

#region ReaderDevice listener implementations

        // 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
        public void OnAvailabilityChanged(ReaderDevice reader)
        {
            if (reader.GetAvailability() == Availability.Available)
            {
                ConnectToReaderDevice();
            }
            else if (reader.GetAvailability() == Availability.Unavailable)
            {
                AlertDialog.Builder alert = new AlertDialog.Builder(this);
                alert
```

```
                .SetTitle("Device became unavailable")
                .SetPositiveButton("OK", (sender, e) => { })
                .Create()
                .Show();
        }
    }

    // The connect method has completed, here you can see whether there was an error with establishing the connection c
    public void OnConnectionCompleted(ReaderDevice readerDevice, Throwable error)
    {
        // If we have valid connection error param will be null,
        // otherwise here is error that inform us about issue that we have while connecting to reader device
        if (error != null)
        {

            // ask for Camera Permission if necessary
            if (error is CameraPermissionException)
                ActivityCompat.RequestPermissions(((ScannerActivity)this), new string[] { Android.Manifest.Permission.C

            UpdateUIByConnectionState();
        }
    }

    // This is called when a connection with the readerDevice has been changed.
    // The readerDevice is usable only in the "ConnectionState.Connected" state
    public void OnConnectionStateChanged(ReaderDevice reader)
    {
        ClearResult();
        if (reader.ConnectionState == ConnectionState.Connected)
        {
            // We just connected, so now configure the device how we want it
            ConfigureReaderDevice();
        }

        isScanning = false;
        UpdateUIByConnectionState();
    }

    // This is called after scanning has completed, either by detecting a barcode, canceling the scan by using the on-s
    public void OnReadResultReceived(ReaderDevice readerDevice, ReadResults results)
    {
        ClearResult();

        if (results.SubResults != null && results.SubResults.Count > 0)
        {
            foreach (ReadResult subResult in results.SubResults)
            {
                CreateResultItem(subResult);
            }
        }
        else if (results.Count > 0)
        {
            CreateResultItem(results.GetResultAt(0));
        }

        isScanning = false;
        btnScan.Text = "START SCANNING";
        resultListAdapter.NotifyDataSetChanged();
    }

    #endregion

..
```

3. Instantiate a *ReaderDevice* object.

Using the MX Reader

Initialize a Reader Device object for MX readers using the following factory method:

```
//*********************************************************************************
// Create an MX-1xxx reader  (note that no license key in needed)
//*********************************************************************************
readerDevice = ReaderDevice.GetMXDevice(this);

//Listen when a MX device has became available/unavailable
if (!availabilityListenerStarted)
{
     readerDevice.StartAvailabilityListening();
     availabilityListenerStarted = true;
}
```

The availability of the MX mobile terminal can change when the device turns on or off, or if the USB cable gets connected or disconnected. You can handle those changes using the following *ReaderDeviceListener* interface method:

```
public void OnAvailabilityChanged(ReaderDevice reader)
```

Using the Camera Reader

You are recommended to use an MX mobile terminal to scan barcodes. However, cmbSDK also supports using the built-in camera of a mobile device. This includes the support of optional external aimers or illumination, and the customization of the live-stream preview's appearance.

To scan barcodes using the built-in camera of a mobile device, initialize the *ReaderDevice* object using the *getPhoneCameraDevice* static method. The camera reader has several options when initialized. The following parameters are required:

- *Context*
- *CameraMode*
- *PreviewOption*
- *ViewGroup*
- *RegistrationKey*
- *CustomData*

The *Context* parameter provides a reference to the activity you are currently in.

The *CameraMode* parameter is of type CameraMode defined in **CameraMode.java** and it accepts one of the values listed in the following table.

These modes provide the following default settings for the reader:

- The zoom feature is available and a button to control it is visible on the live-stream preview (if displayed).
- The simulated hardware trigger (volume control buttons) is disabled.
- When *StartScanning()* is called, the decoding process is started.

Based on the selected mode, additional illumination options and behaviors are set, also listed in the table.

VALUE	DESCRIPTION	ILLUMINATION	LIVE-STREAM PREVIEW
NO_AIMER	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 if the mobile device does not have an aiming accessory.	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 are ignored.	Displayed

VALUE	DESCRIPTION	ILLUMINATION	LIVE-STREAM PREVIEW
PASSIVE_AIMER	Initializes the reader to use a passive aimer. No live-stream preview is available on the device screen in this mode, since an aiming pattern is projected.	Illumination is not available, and the live-stream preview does not have an illumination button.	Not Displayed
		If commands are sent to the reader for illumination control, they are ignored because it is assumed in this mode that the built-in LED of the mobile device is being used for the aimer.	
FRONT_CAMERA	Initializes the reader to use the front camera of the mobile device, if available. Use this configuration with care because most front facing cameras do not have auto focus and illumination, and provide significantly lower resolution images. Illumination is not available in this mode.	The front camera is used.	Displayed
		Illumination is not available and the live-stream preview does not have an illumination button.	
		If commands are sent to the reader for aimer or illumination control, they are ignored.	

The *PreviewOption* parameter is of type *PreviewOption* defined in **PreviewOption.java**, and is used to change the reader's default values or override defaults derived from the selected *CameraMode*. You can specify the following options:

VALUE	DESCRIPTION
DEFAULTS	Accept all defaults set by the CameraMode.
NO_ZOOM_BUTTON	Hides the zoom button on the live-stream preview, preventing the user from adjusting the zoom of the mobile device camera.
NO_ILLUMINATION_BUTTON	Hides the illumination button on the live-stream preview, preventing the user from toggling the illumination.
HARDWARE_TRIGGER	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 purpose-built scanner's trigger, and pressing it a second time does not stop the scanning process.
PAUSED	If using a live-stream preview, the preview is displayed when the *startScanning()* method is called, but the reader does not start decoding until the user presses the on-screen button to start the scanning process.

VALUE	DESCRIPTION
ALWAYS_SHOW	Forces a live-stream preview to be displayed even if an aiming mode is selected (for example CameraMode == PASSIVE_AIMER).
HIGH_RESOLUTION	Uses the device camera in higher resolution, changing the default 1280x720 resolution to 1920x1080 on devices that support it, and to the default resolution on devices that do not support it. This can help with scanning small barcodes, but increases the decoding time as there is more data to process in each frame.
HIGH_FRAME_RATE	Uses the device's camera in 60 FPS instead of the default 30 FPS to provide a smoother camera preview.
SHOW_CLOSE_BUTTON	Show close button in partial view.

The *ViewGroup (optional)* parameter specifies the container for the live-stream preview. If the parameter is left **null**, a full screen preview is used.

The *RegistrationKey (optional)* parameter is used to license your SDK with license key that you have

The CustomData *(optional)* parameter is used for custom tracking

Example

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

```
readerDevice = ReaderDevice.GetPhoneCameraDevice(this, CameraMode.NoAimer, PreviewOption.NoZoomButton | PreviewOption.Pause
```

This starts a preview with the scanner paused and a soft trigger button to toggle scanning. After pressing the soft trigger button, the expected preview look is this:

The viewfinder in the image has an active scanning surface as a result of having set active symbologies. For more details, see Enabling Symbologies.

Requesting Camera Permission for Phone Camera Scanner

From Android 6.0 and above you need to request permission from the user to access the built-in camera of the mobile device.

If the camera cannot be opened due to permission issues, the *onConnectionCompleted(readerDevice, error)* callback contains a *CameraPermissionException* in the error parameter. You can check for this exception type with the *instanceof* operator and request permission within the Activity.

```
if (error is CameraPermissionException)
      ActivityCompat.RequestPermissions(((ScannerActivity) this), new string[] { Android.Manifest.Permission.Camera }, REQU
```

You need to implement the *ActivityCompat.IOnRequestPermissionResultCallback* interface in your Activity to catch the user permission result. To handle user response in *onRequestPermissionsResult(…)*, you can use the following code to retry connecting to the phone camera:

```
       public override void OnRequestPermissionsResult(int requestCode, string[] permissions, [GeneratedEnum] Permission[]
        {
            // Check result from permission request. If it is allowed by the user, connect to readerDevice
            if (requestCode == REQUEST_PERMISSION_CODE)
            {
                if (grantResults.Length > 0 && grantResults[0] == Permission.Granted)
                {
                    if (readerDevice != null && readerDevice.ConnectionState != ConnectionState.Connected)
                        readerDevice.Connect(this);
                }
                else
                {
                    if (ActivityCompat.ShouldShowRequestPermissionRationale(((ScannerActivity)this), Android.Manifest.Permi
                    {
                        AlertDialog.Builder builder = new AlertDialog.Builder(this)
                                .SetMessage("You need to allow access to the Camera")
                                .SetPositiveButton("OK", (sender, e) =>
```

```
                            {
                                ActivityCompat.RequestPermissions(this, new string[] { Android.Manifest.Permission.Came
                            })
                            .SetNegativeButton("Cancel", (sender, e) => { });
                    AlertDialog dialog = builder.Create();
                    dialog.Show();
                }
            }
        }
    }
```

Connecting to the Reader Device

Before connecting, set the *IReaderDeviceListener* object to receive events:

```
readerDevice.SetReaderDeviceListener(this);
```

Additionally, you can enable sending the last triggered image and SVG from the reader:

```
readerDevice.EnableImage(true);
readerDevice.EnableImageGraphics(true);
```

Invoke the connect method after initializing the *ReaderDevice* and setting a listener method to handle responses from the reader. The connect method takes *IOnConnectionCompletedListener* as parameter:

```
 //Make sure the device is turned ON and ready
readerDevice.Connect(ScannerActivity.this);
```

The following listener methods are called with the new *ReaderDevice* status information:

```
public void onConnectionStateChanged(ReaderDevice reader);
public void onConnectionCompleted(ReaderDevice reader, Throwable err)
```

The *onConnectionCompleted* method passed as a parameter of connect is also invoked as the connection process completes. If there was a connection error, this method provides a *Throwable* object.

Scanning Barcodes

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

You can start scanning barcodes with a properly configured reader by calling the *StartScanning* method from your *ReaderDevice* class:

```
readerDevice.StartScanning();
```

- If using an MX mobile terminal, you can press a trigger button on the device to turn the scanner on and read a barcode.
- If using the camera reader, 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.

You can stop scanning with the following:

```
readerDevice.StopScanning();
```

Scanning stops under one of the following conditions:

- The reader found and decoded a barcode.
- You 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 calls the *StopScanning()* method.

When a barcode is decoded successfully, you receive a ReadResults iterable result collection object in the *ReaderDevice* listener method. The *onReadResultReceived* listener method is invoked either because the reader decoded a barcode or the scanning process was complete.

Example

```
        // This is called after scanning has completed, either by detecting a barcode, canceling the scan by using the on-s
        public void OnReadResultReceived(ReaderDevice readerDevice, ReadResults results)
        {
            ClearResult();

            if (results.SubResults != null && results.SubResults.Count > 0)
            {
                foreach (ReadResult subResult in results.SubResults)
                {
                    CreateResultItem(subResult);
                }
            }
            else if (results.Count > 0)
            {
                CreateResultItem(results.GetResultAt(0));
            }

            isScanning = false;
            btnScan.Text = "START SCANNING";
            resultListAdapter.NotifyDataSetChanged();
        }
```

Enabling Symbologies

CmbSDK does not enable any *symbologies* by default for barcode reading with the built-in camera of the mobile device. You must enable all barcode symbologies your application needs to scan to achieve optimal scanning performance.

Individual symbologies can be enabled using the following method of the *ReaderDevice* class:

```
public void SetSymbologyEnabled(final Symbology symbology, final boolean enable, final IOnSymbologyListener listener)
readerDevice.SetSymbologyEnabled(Symbology.Datamatrix, true, null);
readerDevice.SetSymbologyEnabled(Symbology.UpcEan, true, null);
```

All symbologies used for the symbology parameter in this method can be found in **ReaderDevice.java**.

Examples

```
 /* Enable QR scanning */
readerDevice.SetSymbologyEnabled(Symbology.Qr, true, null);
```

You can also use the same method to disable symbologies:

```
/ * Disable Code 25 scanning */ readerDevice.SetSymbologyEnabled(Symbology.C25, false, null);
```

You can implement the method for *IOnSymbologiesListener* to check the result of the symbology change:

```
public void onSymbologyEnabled(ReaderDevice reader, Symbology symbology, Java.Lang.Boolean enabled, Throwable error) {
if (error != null) {
/* Unsuccessful
probably the symbology is unsupported by the current device, or there is a problem with the connection between the readerDe
} else {
// Success }
}
```

Illumination Control

If your reader device is equipped with illumination lights, you can control them: when scanning starts, you can turn them on or off. Use the following method of your Reader Device object:

```
readerDevice.SetLightsOn(true, null);
```

You can implement the interface method for *IOnLightsListener*, which is the second parameter of the method.

```
public class ScannerActivity : AppCompatActivity, .... IOnLightsListener .... { ....

public void onLightsOnCompleted(ReaderDevice reader, Java.Lang.Boolean on, Throwable error) {
if (error != null) { // Unsuccessful
} else {
// Success
}}
```

Not all devices and device modes support illumination control.

Camera Zoom Settings

If the built-in camera of a mobile device is used as the reader device, you can configure zoom levels and how they are used. There are three zoom levels:

- normal: not zoomed (100%)
- level 1 zoom (150% on Android by default)
- level 2 zoom (300% on Android by default)

The *SET CAMERA.ZOOM-PERCENT [100-MAX] [100-MAX]* command is for configuring how far the two levels zoom in percentage. 100 is not zoomed and MAX (goes up to 1000) zooms as far as the device is capable of. The first argument is used for setting level 1 zoom, and the second for level 2 zoom.

You can check the current zoom setting with the *GET CAMERA.ZOOM-PERCENT* command, which returns two values: level 1 and level 2 zoom.

Example

```
readerDevice.DataManSystem.SendCommand("SET CAMERA.ZOOM-PERCENT 250 500");
```

Note: The camera needs to be started within cmbSDK 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 only go up to 600, the *GET CAMERA.ZOOM-PERCENT* command returns 1000 as long as camera is not opened, but it returns 600 afterwards.

GET/SET CAMERA.ZOOM 0-2 is another command that sets the zoom level or returns the actual setting. Possible values for the SET command are:

- 0 - normal (not zoomed)
- 1 - level 1 zoom
- 2 - level 2 zoom

You can call this command before or even during scanning, and the zoom goes up to the configured level. If scanning is finished, the value is reset to normal behavior (0).

Example

```
readerDevice.DataManSystem.SendCommand("SET CAMERA.ZOOM 2");
```

Camera Overlay Customization

When using the mobile device's camera, cmbSDK allows you to see the camera preview inside a preview container or in full screen. This preview also contains a customizable overlay. The cmbSDK camera overlay features buttons for zooming, flashing and closing the scanner, and a progress bar indicating the scan timeout.

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

```
MWOverlay.OverlayMode = MWOverlay.OverlayModeEnum.OmLegacy;
```

The customization of the legacy camera overlay is limited, so it is recommended to use the cmbSDK overlay.

When using the cmbSDK overlay:

1. Copy the layout files from the Resources/layout directory into your project and modify them. Use **cmb_scanner_partial_view.xml** if scanning is started inside a container (partial view), and use **cmb_scanner_view.xml** if scanning is started in full screen.
2. Modify the layout according to your needs. For example, you can change the sizes, positions or color of the views, remove views and add your own views, like an overlay image.

CmbSDK accesses the views it uses (zoom, flash, close buttons, the view used for drawing lines on the corners, and the progress bar) with the android:tag attribute. Do not change the android:tag attribute, otherwise cmbSDK cannot recognize the views and continues to function as if they are removed.

Both the cmbSDK and the legacy overlay allow you to change the images used on the zoom and flash buttons if your images have the same name as the names cmbSDK uses. You can find the images and names used in cmbSDK in the Resources/**drawable-mdpi** and **drawable-hdpi** directories. While the other resolutions are optional, these two directories must contain your images with the correct names so that cmbSDK displays the proper images.

Both the cmbSDK and the legacy overlay allow you to change the color and width of the rectangle that is displayed when a barcode is detected.

Example:

```
MWOverlay.LocationLineColor = Color.Yellow;
MWOverlay.LocationLineWidth = 6;
```

Advanced Configuration using DataMan Control Commands

Cognex scanning devices implement DataMan Control Commands (DMCC) for configuring and controlling the device. Every feature of the device can be controlled using this text-based language. The API provides a method for sending DMC commands to the device. Commands exist both for setting and querying configuration properties. DMC commands are same for all platforms and frameworks.

The Appendix includes the complete DMCC reference for the camera reader.

> The DMCCs for MX mobile terminals and other supported devices can be found in their respective manuals available through Setup Tool.

The following examples show different DMCC sent to the device for more advanced configuration.

Examples

```
//Change the scan direction to omnidirectional
readerDevice.DataManSystem.SendCommand("SET DECODER.1D-SYMBOLORIENTATION 0", ScannerActivity.this);
//Change live-stream preview's scanning timeout to 10 seconds
readerDevice.DataManSystem.SendCommand("SET DECODER.MAX-SCAN-TIMEOUT 10", ScannerActivity.this);
```

You can also invoke DMCC query commands and receive their response in the *IOnResponseReceivedListener.onResponseReceived()* method.

```
//Get the type of device connected readerDevice.DataManSystem.SendCommand("GET DEVICE.NAME", this);

public void onResponseReceived(DataManSystem dataManSystem, DmccResponse dmccResponse) {
if (dmccResponse.Error != null) {
// Unsuccessful
Log.e("DMCC_ERR", “GET DEVICE.NAME failed”, dmccResponse.Error.Message);
} else {
// Success - Use the following result fields:
//int mResponseId = dmccResponse.ResponseId;
//String mPayLoad = dmccResponse.PayLoad;
//byte[] mBinaryData = dmccResponse.GetBinaryData();
} }
```

Resetting the Configuration

> NOTE: This section includes resetting to CmbSDK defaults and does not include instruction on resetting to factory defaults.

CmbSDK includes a method for resetting the device to its default settings. In case of an MX mobile terminal, the default setting are the saved configurations. In case of a built-in camera, the default settings are the defaults identified in the Appendix, where no symbologies are enabled.

To reset the device, add:

```
readerDevice.ResetConfig(null);
```

When using an MX mobile terminal, there are three states that we can distinguish:

- Factory defaults
- Saved configuration: when there were different configurations set on the device and CONFIG.SAVE DMCC was called.
- Session configuration: when you make changes on the saved configuration, the changes are valid until the MX Mobile Terminal is rebooted. If it is rebooted, it has the saved configuration state.

You can monitor the completion of this async method using the *IOnResetConfigListener* interface, which is an optional parameter.

```
public class ScannerActivity : AppCompatActivity, .... IOnResetConfigListener .... { ....

public void onResetConfigCompleted(ReaderDevice reader, Throwable error) {
if (error != null) { // Unsuccessful
} else {
// Success
}
```

Working with Results

When a barcode is successfully read, the *onReadResultReceived* method creates and returns a ReadResult object. In case of having multiple barcodes successfully read on a single image or frame, multiple *ReadResult* objects are returned in the *ReadResult* object.

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

- **IsGoodRead** (bool): tells whether the read was successful or not
- **ReadString** (string): the decoded barcode as a string
- **Image** (Bitmap): the image/frame that the decoder processed
- **ImageGraphics** (string): the boundary path of the barcode as SVG data
- **Xml** (string): the raw XML that the decoder returned
- **Symbology** (Symbology): the symbology type of the barcode. This enum is defined in ***ReaderDevice.java.***

When a scanning ends with no successful read, a *ReadResult* is returned with the *IsGoodRead* property set to false.

To enable the Image and ImageGraphics properties being filled in the *ReadResult* object, set the corresponding *EnableImage()* and/or *EnableImageGraphics()* properties of the *ReaderDevice* object.

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.

Image Results

The image and SVG results are disabled by default, which means that when scanning, the *ReadResults* do not contain any data in the corresponding properties.

To enable image results, invoke the *EnableImage()* method from the *ReaderDevice* object:

```
readerDevice.EnableImage(true);
```

To enable SVG results, invoke the *EnableImageGraphics()* method on *ReaderDevice* object:

```
readerDevice.EnableImageGraphics(true);
```

Handling Disconnects

If a device disconnects due to low battery condition or manual cable disconnection, it can be detected by the *onConnectionStateChanged()* method of the *IReaderDeviceListener* interface.

> **Note:** The *onAvailabilityChanged()* method of *IReaderDeviceListener* is also called when the device becomes physically unavailable. It means that (re)connection is not possible. Always check the *GetAvailability()* method of the *ReaderDevice* object before trying to call the *Connect()* method.

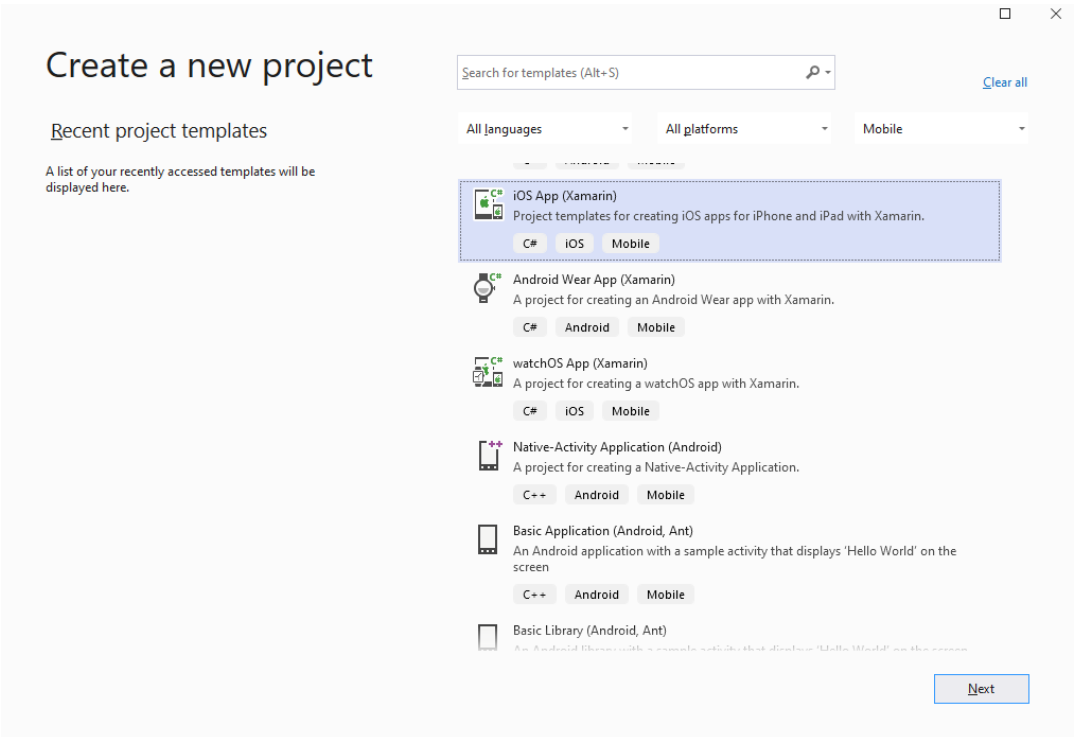
Xamarin.iOS

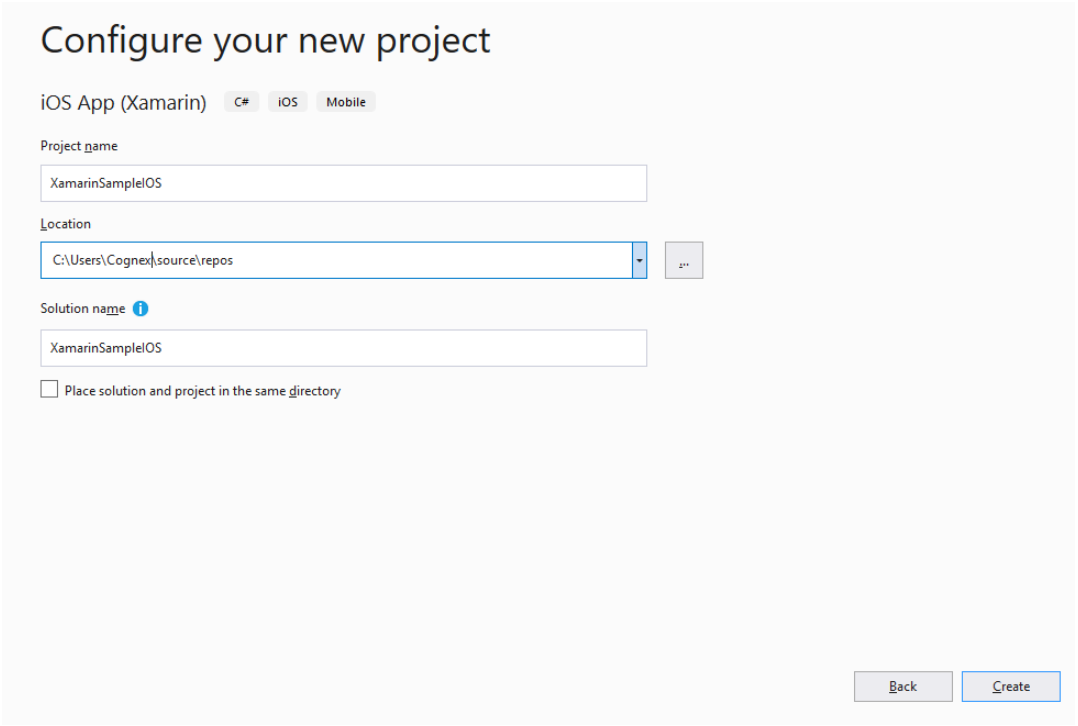
Getting Started

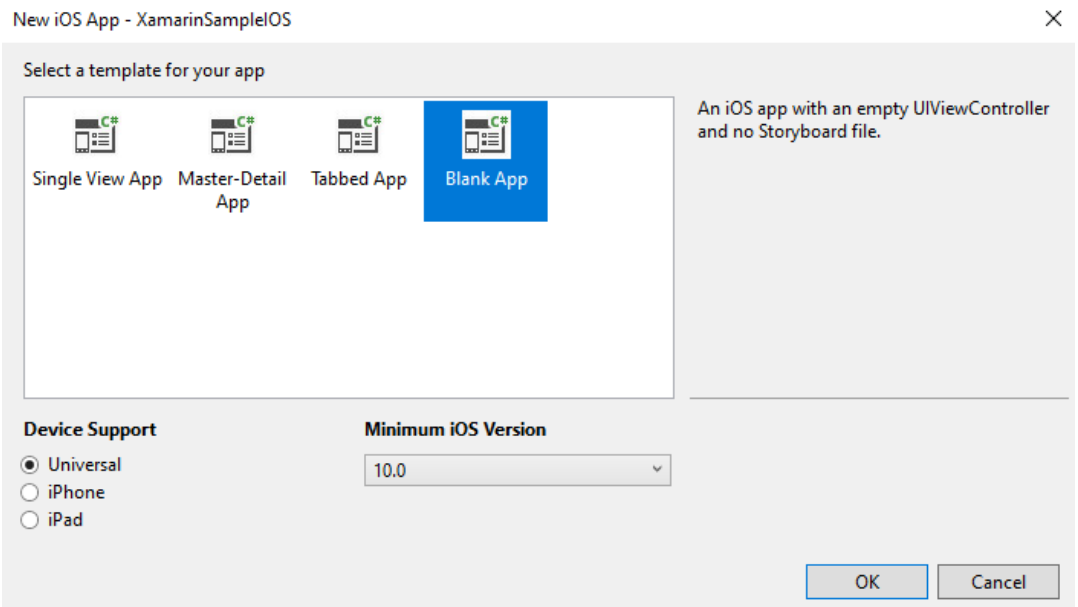
Open Visual Studio and follow this steps:

1. Go to **File -> New -> Project.**

2. Select **iOS App (Xamarin) ,** set project name and from templates select **Blank App**

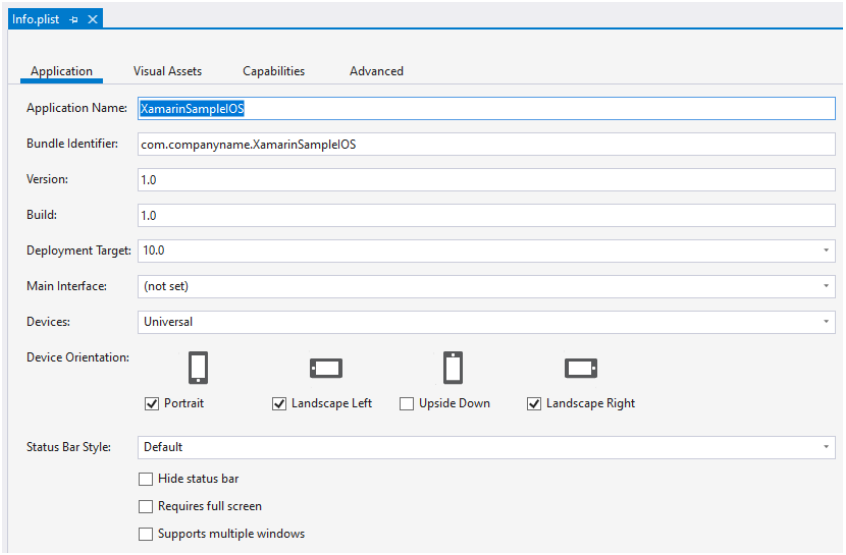






After creating a blank application, create all resources you will use (images, storyboards, controllers, etc.). You can copy them from our sample.

Next open your **Info.plist** file and set some project properties for your needs (app name, deployment target, main interface, etc..).



Important thing here is to add **Camera permission** for this app. In Visual Studio there is no options to add this permission from here. You need to open your **Info.plist** file in some text editor and add these lines:

```
<key>NSCameraUsageDescription</key>
<string>Camera used for scanning</string>
```

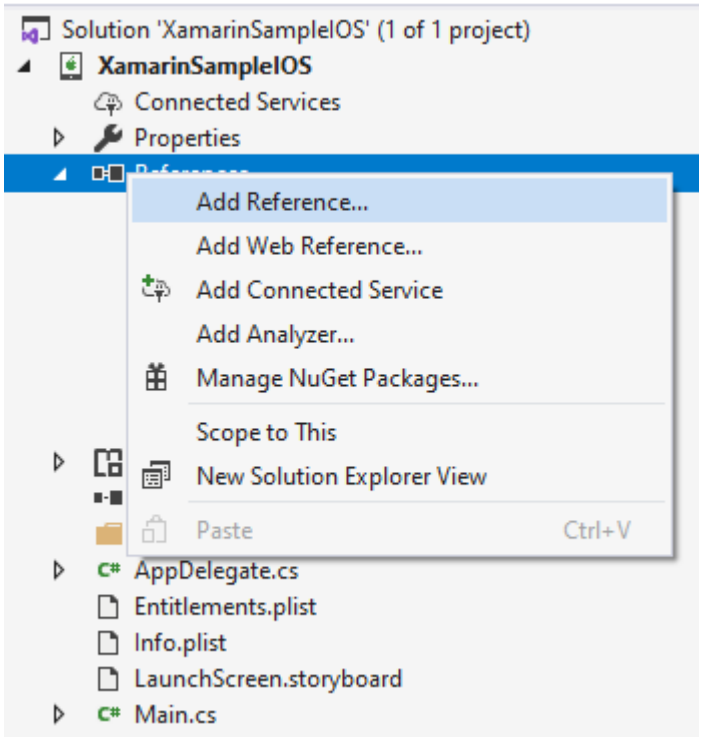
If you use MX Device as reader device add this protocols in Info.plist:

```
<key>UISupportedExternalAccessoryProtocols</key>
<array>
  <string>com.cognex.dmcc</string>
  <string>com.demo.data</string>
</array>
```

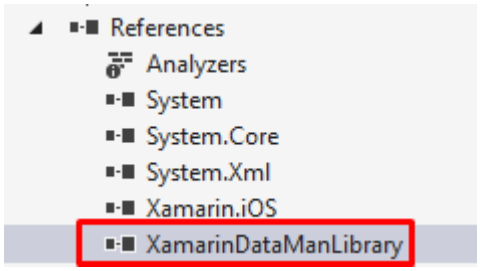
and please check this section before uploading your app on the App Store: Getting your MX Mobile Terminal Enabled App into the App Store

Reference cmbSDK

Now we need to reference **XamarinDataManLibrary.dll** in order to use the cmbSDK.



Browse and find **XamarinDataManLibrary.dll** file and add it as a reference in this project.



If you plan to use your application on devices with iOS lower than 12.2 extra steps are required:

- Add Swift dylib dependencies for runtime support. Starting from iOS 12.2 and Swift 5.1, the language became ABI (application binary interface) stable and compatible. That's why any application targeting a lower iOS version needs to include Swift dylibs dependencies used by the framework. Use the SwiftRuntimeSupport NuGet package to automatically include required dylib dependencies into the resulting application package.
- Add **SwiftSupport** folder with signed dylibs, which is validated by the AppStore during the uploading process. The package should be signed and distributed to the AppStore connect using Xcode tools, otherwise, it will be automatically rejected.

Note: This package works only on real devices. You won't be able to test your application on simulators with an iOS version lower than 12.2.

Licensing the SDK

To use cmbSDK for barcode scanning with a mobile device without an MX mobile terminal, you need to install a license key. If the license key is missing, asterisks will appear instead of scanned results.

Contact your Cognex Sales Representative for information on how to obtain a license key, including 30-day trial licenses.

After obtaining your license key there is two ways to add your license key in application.

The first one is to implement an activation directly from the code when you create your readerDevice:

```
//*****************************************************************************
// Create a camera reader (for either the built-in camera or an MX-100)
//
```

```
// NOTE: if we are connecting to a MX-100 (cameraMode == kCDMCameraModeActiveAimer) then
//       no license key is needed. However, if we're scanning using the built-in camera
//       of the mobile phone or tablet, then the SDK requires a license key. Refer to
//       the SDK's documentation on obtaining a license key as well as the methods for
//       passing the key to the SDK (in this example, we're relying on an entry in
//       plist.info--there are also readerOfDeviceCamera methods where it can be passed
//       as a parameter).
//*********************************************************************************
case DataManDeviceClass.PhoneCamera:
     readerDevice = CMBReaderDevice.ReaderOfDeviceCameraWithCameraMode(cameraMode, CDMPreviewOption.Defaults, null, "SDK_KE
     break;
```

and second is to add it as a key with a value in the project specific **Info.plist** file:

```
<key>MX_MOBILE_LICENSE</key>
<string>Your license key</string>
```

Implementing SDK

1. First build your UI 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**.

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

```
//--------------------------------------------------------------------------
        // 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(NSNotification obj)
        {
            if (readerDevice != null && readerDevice.Availability == CMBReaderAvailibility.Available && readerDevice.Co
            {
                ConnectToReaderDevice();
            }
        }

        public override void ViewDidLoad()
        {
            base.ViewDidLoad();

            // Add our observer for when the app becomes active (to reconnect if necessary)
            NSNotificationCenter.DefaultCenter.AddObserver(UIApplication.DidBecomeActiveNotification, AppBecameActive);
        }


// 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
        [Export("availabilityDidChangeOfReader:")]
        public void AvailabilityDidChangeOfReader(CMBReaderDevice reader)
        {
            ClearResult();

            if (reader.Availability != CMBReaderAvailibility.Available)
            {
                ShowAlert(null, "Device became unavailable");
            }
            else
            {
                ConnectToReaderDevice();
            }
        }

        // This is called when a connection with the self.readerDevice has been changed.
        // The self.readerDevice is usable only in the "CMBConnectionStateConnected" state
        [Export("connectionStateDidChangeOfReader:")]
        public void ConnectionStateDidChangeOfReader(CMBReaderDevice reader)
        {
            isScanning = false;
            ClearResult();

            if (readerDevice.ConnectionState == CMBConnectionState.Connected)
            {
                // We just connected, so now configure the device how we want it
                ConfigureReaderDevice();

                if (connectingAlert != null)
                {
                    connectingAlert.DismissViewController(true, () =>
                    {
                        connectingAlert = null;
                    });
                }
            }
            else if (readerDevice.ConnectionState == CMBConnectionState.Disconnected && connectingAlert != null)
            {
                connectingAlert.DismissViewController(true, () =>
                {
                    connectingAlert = null;
                });
            }

            UpdateUIByConnectionState();
        }

        // This is called after scanning has completed, either by detecting a barcode, canceling the scan by using the
        [Export("didReceiveReadResultFromReader:results:")]
        public void DidReceiveReadResultFromReader(CMBReaderDevice reader, CMBReadResults readResults)
        {
            isScanning = false;
            btnScan.Selected = false;

            scanResults.RemoveAllObjects();

            if (readResults.SubReadResults != null && readResults.SubReadResults.Length > 0)
            {
                scanResults.AddObjects(readResults.SubReadResults);
            }
            else if (readResults.ReadResults.Length > 0)
            {
                scanResults.Add(readResults.ReadResults[0]);
            }

            tableViewSource.SetItems(scanResults);
            tvResults.ReloadData();
        }
```

3. 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):

```
readerDevice = CMBReaderDevice.ReaderOfMXDevice();
readerDevice.WeakDelegate = this;
```

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.

```
public 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. 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*, and it accepts one of the following values:

- **NoAimer**: 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.
- **PassiveAimer**: 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.
- **ActiveAimer**: 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.
- **FrontCamera**: 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 *CDMPreviewOption.Paused* for more details).

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

- **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.

- **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.

- **ActiveAimer** (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.

- **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*) 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:

- **Defaults**: Accept all defaults set by the **CameraMode**.
- **NoZoomBtn**: Hide the zoom button on the live-stream preview.
- **NoIllumBtn**: Hide the illumination button on the live-stream preview.
- **HwTrigger**: Enable simulated hardware trigger (volume controls) for starting scanning. When pressed, scanning starts.
- **Paused**: 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.
- **AlwaysShow**: Force display of live-preview when active or passive aiming mode has been selected (e.g. *CameraMode == PassiveAimer*)
- **PessimisticCaching**: Use only when *CameraMode == ActiveAimer*, this will read the settings from the **ActiveAimer** when the app resumes from background, in case the aimer settings were changed from another app.
- **HighResolution**: 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.
- **HighFrameRate**: 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:

```
readerDevice = CMBReaderDevice.ReaderOfDeviceCameraWithCameraMode(CDMCameraMode.NoAimer, CDMPreviewOption.Defaults, null);
readerDevice.WeakDelegate = this;
```

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

```
readerDevice = CMBReaderDevice.ReaderOfDeviceCameraWithCameraMode(CDMCameraMode.NoAimer, CDMPreviewOption.NoZoomBtn | CDMPr
readerDevice.WeakDelegate = this;
```

Connecting to the Device

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

Then connect using **ConnectWithCompletion**:

```
// Make sure the device is turned ON and ready
if (readerDevice.Availability == CMBReaderAvailibility.Available && readerDevice.ConnectionState == CMBConnectionState.Disc
{
    if (readerDevice.DeviceClass == DataManDeviceClass.PhoneCamera && cameraMode == CDMCameraMode.ActiveAimer)
    {
        connectingAlert = UIAlertController.Create("Connecting", null, UIAlertControllerStyle.Alert);
        PresentViewController(connectingAlert, true, null);
    }

    readerDevice.ConnectWithCompletion((error) =>
    {
        if (error != null)
        {
            ShowAlert("Failed to connect", null);
        }
    });
}
```

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 **CMBConnectionState,Connected**, which means that you have successfully made the connection to the *CMBReaderDevice*, and can begin using the Cognex Mobile Barcode SDK.

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 are 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: **PreviewOption.PessimisticCaching**.

Enabling Symbologies

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

```
readerDevice.SetSymbology(CMBSymbology.DataMatrix, true, null);
```

All symbologies used for the symbology parameter in this method can be found in **CMBSymbology** enum.

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

```
readerDevice.SetSymbology(CMBSymbology.DataMatrix, false, null);
```

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:

```
readerDevice.SetLightsON(true, (error) =>
{
```

```
    if (error != null)
    {
        System.Diagnostics.Debug.WriteLine("// Failed to enable illumination, Possible causes are: reader disconnected, out
    }
});
```

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

```
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

```
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.

```
MWOverlay.SetOverlayMode((int) OverlayMode.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 VisualStudio 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:

```
MWOverlay.SetLocationLineUIColor(UIColor.Yellow);
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.

DMC commands are same for all platforms and frameworks.

The following examples show different DMCC commands being sent to the device for more advanced configuration.

Example:

Change the scan direction to omnidirectional:

```
readerDevice.DataManSystem.SendCommand("SET DECODER.1D-SYMBOLORIENTATION 0", (response) =>
{
    if (response.Status == CDMResponseStatus.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:

```
readerDevice.DataManSystem.SendCommand("SET DECODER.MAX-SCAN-TIMEOUT 10", (response) =>
{
    if (response.Status == CDMResponseStatus.DMCC_STATUS_NO_ERROR)
    {
        // Command was executed successfully
    }
    else
    {
        // Command failed, handle errors here
    }
});
```

Get the type of the connected device:

```
readerDevice.DataManSystem.SendCommand("GET DEVICE.TYPE", (response) =>
{
    if (response.Status == CDMResponseStatus.DMCC_STATUS_NO_ERROR)
    {
        // Command was executed successfully
    }
    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:

```
readerDevice.ResetConfigWithCompletion((error) =>
{
    if (error != null)
    {
        // Failed to reset configuration, Possible causes are: reader disconnected, out of battery or 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 **ICMBReaderDeviceDelegate** method:

```
public void DidReceiveReadResultFromReader(CMBReaderDevice reader, CMBReadResults readResults)
```

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

```
public void DidReceiveReadResultFromReader(CMBReaderDevice reader, CMBReadResults readResults)
{
    if (readResults.ReadResults.Length > 0)
    {
        CMBReadResult readResult = (CMBReadResult)readResults.ReadResults[0];

        if(readResult.Image != null)
        {
            ivPreview.Image = readResult.Image;
        }

        if (readResult.ReadString != null)
        {
            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** 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** (string): the decoded barcode as a string
- **Image** (UIImage): the image/frame that the decoder has processed
- **ImageGraphics** (NSData): the boundary path of the barcode as SVG data
- **XML** (NSData): the raw XML that the decoder returned
- **Symbology** (CMBSymbology): the symbology type of the barcode.

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.

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:

```
readerDevice.ImageResultEnabled = false;
```

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

```
readerDevice.SVGResultEnabled = false;
```

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 *ICMBReaderDeviceDelegate*.

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 **ICMBReaderDeviceDelegate** method: *void AvailabilityDidChangeOfReader(CMBReaderDevice reader) so w*hen the reader becomes available, you can connect.

For example:

```
public void AvailabilityDidChangeOfReader(CMBReaderDevice reader)
{
    if (readerDevice.Availability == CMBReaderAvailibility.Available)
    {
        readerDevice.ConnectWithCompletion((error) =>
        {
            if (error != null)
            {
                // 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:

```
public override void ViewDidLoad()
{
    base.ViewDidLoad();

    NSNotificationCenter.DefaultCenter.AddObserver(UIApplication.DidBecomeActiveNotification, AppBecameActive);
}

void AppBecameActive(NSNotification obj)
{
    if (readerDevice != null && readerDevice.Availability == CMBReaderAvailibility.Available && readerDevice.ConnectionStat
    {
        readerDevice.ConnectWithCompletion((error) =>
        {
            if (error != null)
            {
                // handle connection error
            }
        });
    }
}
```

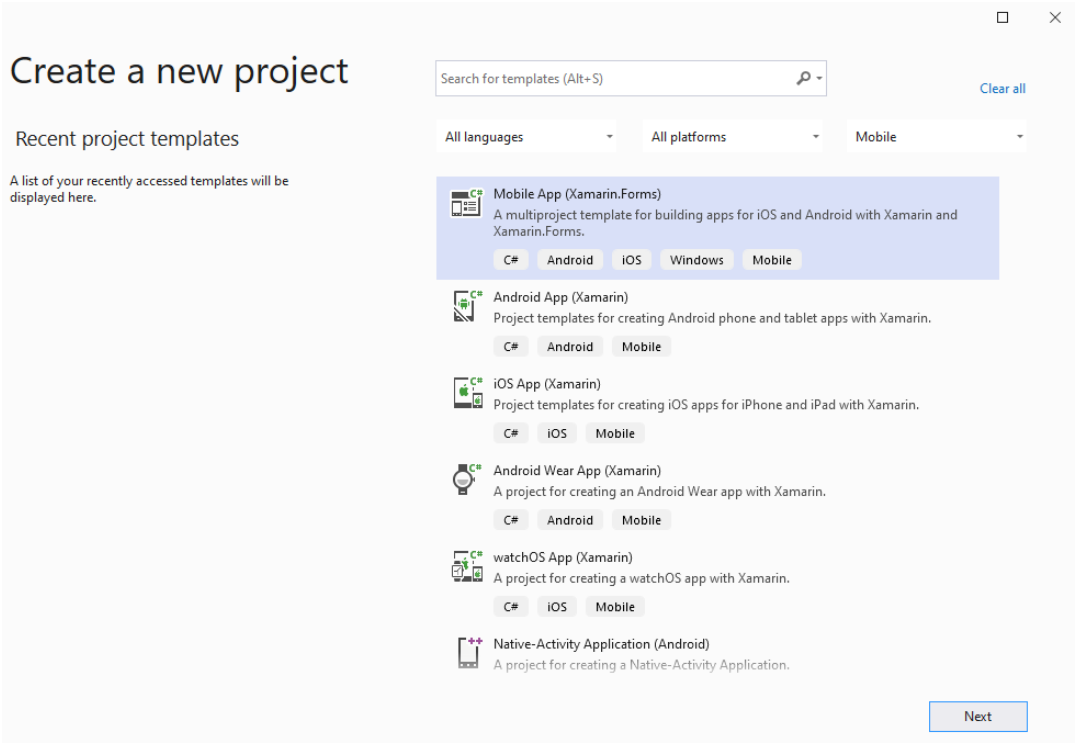
Xamarin.Forms

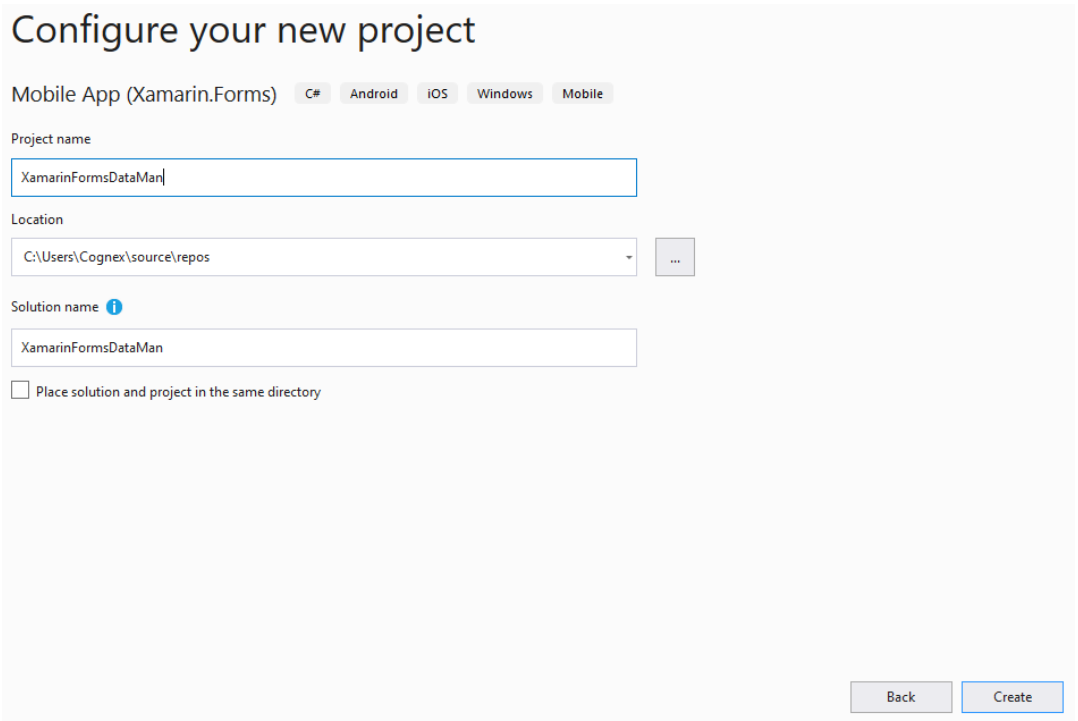
Getting Started

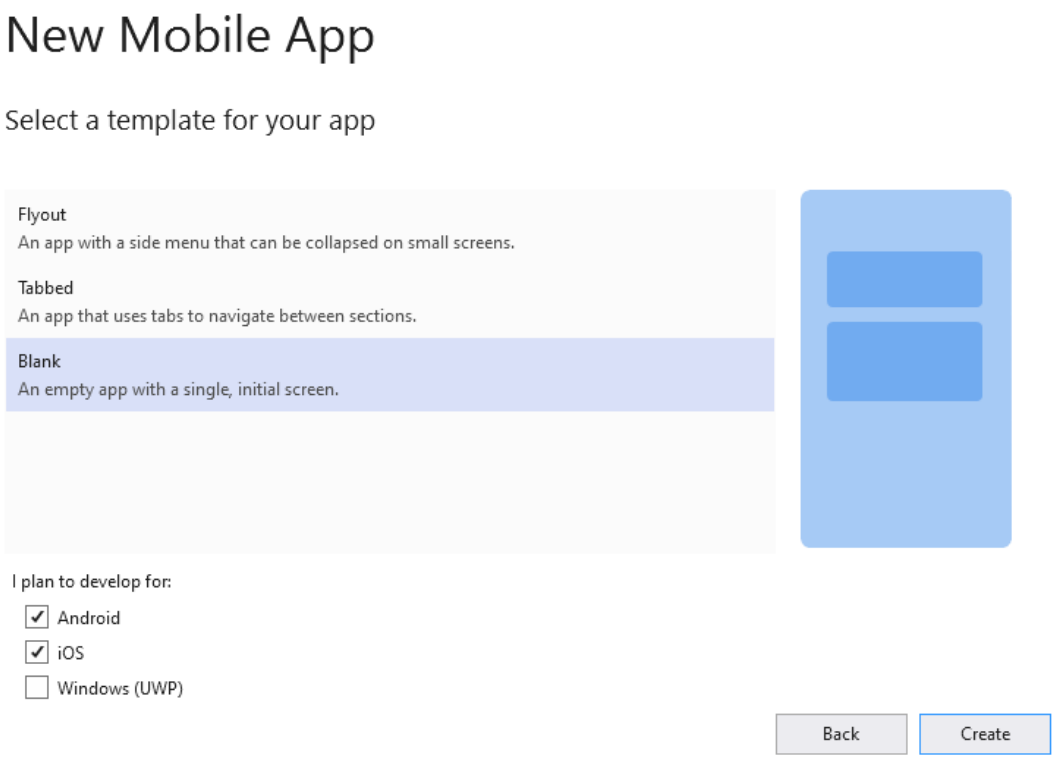
Open Visual Studio and follow these steps:

1. Go to **File -> New -> Project.**

2. Create **Mobile App(Xamarin.Forms) ,** set project name, from templates select **Blank** and choose **Android** and **iOS** platform.

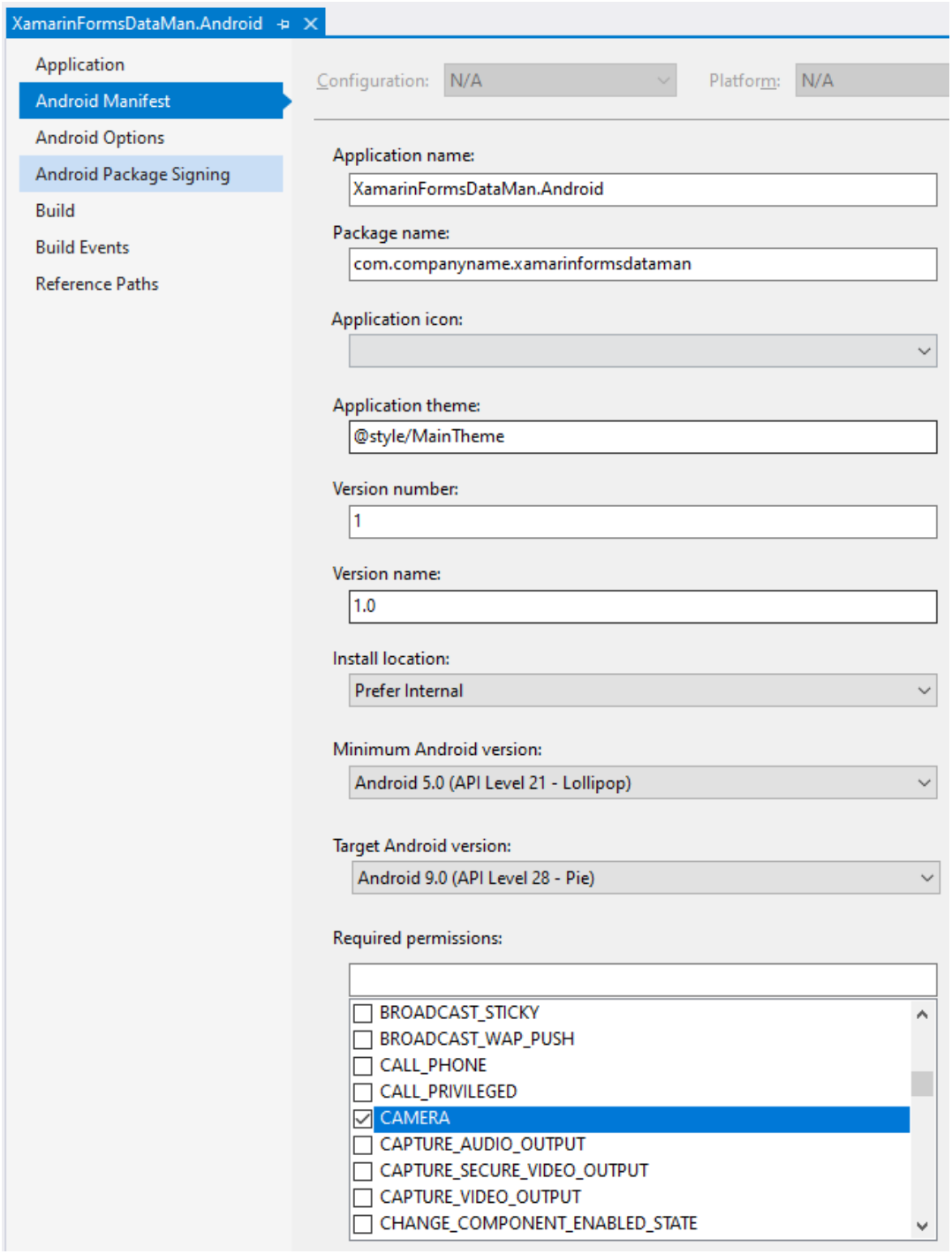

Create a new project
Search for templates (Alt+S)
Clear all
Recent project templates
All languages
All platforms
Mobile
A list of your recently accessed templates will be displayed here.
Mobile App (Xamarin.Forms)
A multiproject template for building apps for iOS and Android with Xamarin and Xamarin.Forms.
C# Android iOS Windows Mobile
Android App (Xamarin)
Project templates for creating Android phone and tablet apps with Xamarin.
C# Android Mobile
iOS App (Xamarin)
Project templates for creating iOS apps for iPhone and iPad with Xamarin.
C# iOS Mobile
Android Wear App (Xamarin)
A project for creating an Android Wear app with Xamarin.
C# Android Mobile
watchOS App (Xamarin)
A project for creating a watchOS app with Xamarin.
C# iOS Mobile
Native-Activity Application (Android)
A project for creating a Native-Activity Application.
Next


Configure your new project
Mobile App (Xamarin.Forms) C# Android iOS Windows Mobile
Project name
XamarinFormsDataMan
Location
C:\Users\Cognex\source\repos
...
Solution name
XamarinFormsDataMan
Place solution and project in the same directory
Back
Create

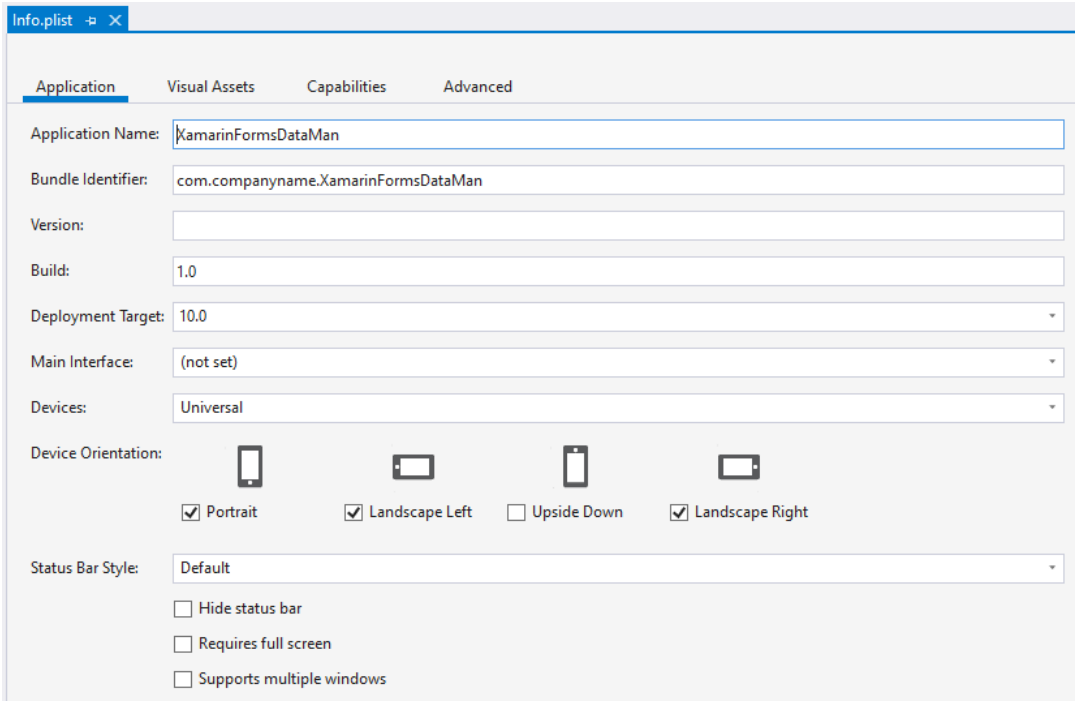


Next right click on your Android platform specific project file, then click Properties and go to Android Manifest section.

Setup your Manifest file (app name, app icon, minimum and maximum android versions), make sure to enable **Camera permission**.



We should do same for iOS platform specific project. Open your **Info.plist** file and set some project properties for your needs (app name, deployment target, etc..)



and make sure to add **Camera permission.** In Visual Studio there is no options to add this permission from here. You need to open your **Info.plist** file in some text editor and add these lines:

```
<key>NSCameraUsageDescription</key>
<string>Camera used for scanning</string>
```

If you use MX Device as reader device add this protocols in Info.plist:

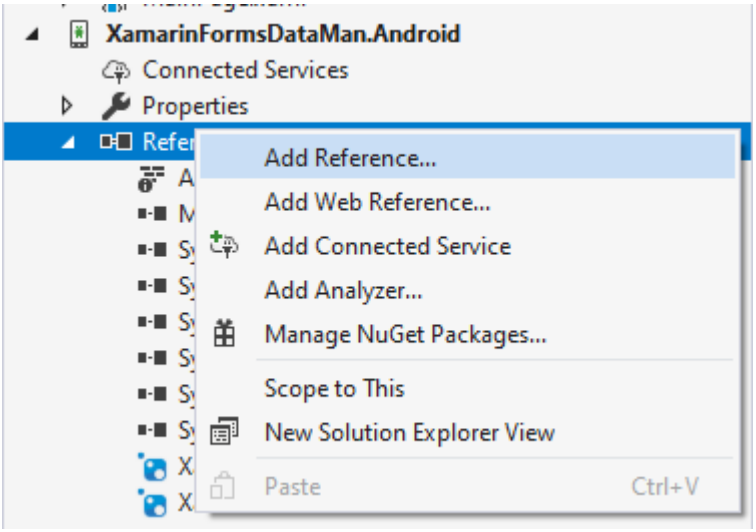
```
<key>UISupportedExternalAccessoryProtocols</key>
<array>
   <string>com.cognex.dmcc</string>
   <string>com.demo.data</string>
</array>
```

and please check this section before uploading your app on the App Store: Getting your MX Mobile Terminal Enabled App into the App Store

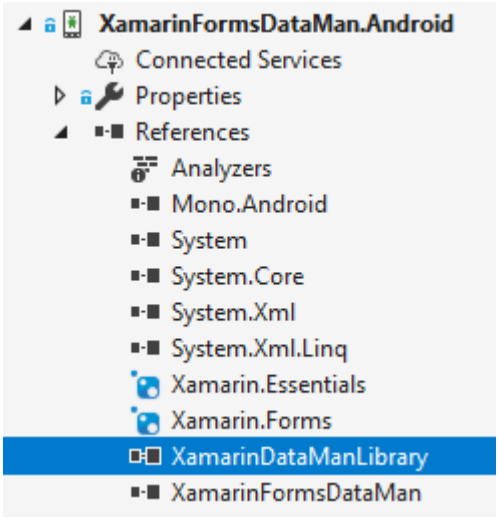
Reference cmbSDK

Now we need to reference **XamarinDataManLibrary.dll** in both platform-specific projects in order to use the cmbSDK.

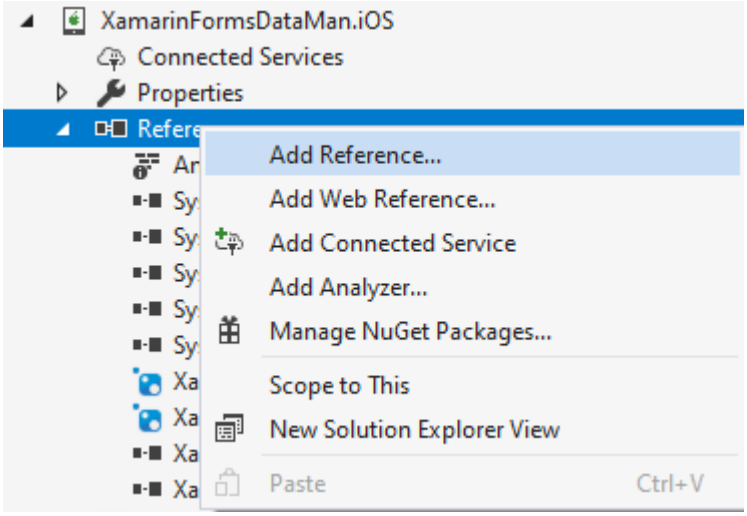
First, add that reference in the Android platform-specific project:

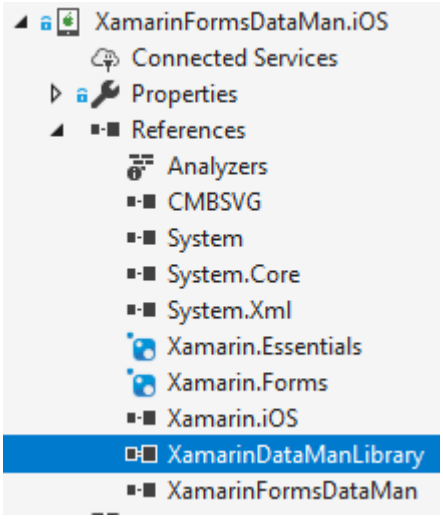


Browse and find **XamarinDataManLibrary.dll** file and add it as a reference in this project.



Next do the same for iOS:





If you plan to use your application on devices with iOS lower than 12.2 extra steps are required:

- Add Swift dylib dependencies for runtime support. Starting from iOS 12.2 and Swift 5.1, the language became ABI (application binary interface) stable and compatible. That's why any application targeting a lower iOS version needs to include Swift dylibs dependencies used by the framework. Use the SwiftRuntimeSupport NuGet package to automatically include required dylib dependencies into the resulting application package.
- Add **SwiftSupport** folder with signed dylibs, which is validated by the AppStore during the uploading process. The package should to be signed and distributed to the AppStore connect using Xcode tools, otherwise, it will be automatically rejected.

Note: This package works only on real devices. You won't be able to test your application on simulators with an iOS version lower than 12.2.

Licensing the SDK

To use cmbSDK for barcode scanning with a mobile device without an MX mobile terminal, you need to install a license key. If the license key is missing, asterisks will appear instead of scanned results.

Contact your Cognex Sales Representative for information on how to obtain a license key, including 30-day trial licenses.

If you obtain cross platform license (one license for both platforms) implement an activation directly from the code when you create camera scanner:

```
if (param_deviceClass == ScannerDevice.PhoneCamera)
{
    //*************************************************************************************
    // Create a camera scanner (for either the built-in camera or an MX-100)
    //
    // NOTE: if we are connecting to a MX-100 (cameraMode == ScannerCameraMode.ActiveAimer) then
    //       no license key is needed. However, if we're scanning using the built-in camera
    //       of the mobile phone or tablet, then the SDK requires a license key. Refer to
    //       the SDK's documentation on obtaining a license key as well as the methods for
    //       passing the key to the SDK (in this example, we're relying on an entry in
    //       plist.info and androidmanifest.xml--there are also DeviceCamera methods where it can be passed
    //       as a parameter).
    //*************************************************************************************
    scannerControl.GetPhoneCameraDevice(param_cameraMode, ScannerPreviewOption.Defaults, true, "SDK_KEY");
}
```

Otherwise if you have different keys (key for iOS only and key for Android only) add these keys inside AndroidManifest.xml or Info.plist files.

For Android add the following line in the AndroidManifest.xml file of your application under the application tag:

```
<application>
        .....
     <meta-data android:name="MX_MOBILE_LICENSE"
```

```
          android:value="YOUR_MX_MOBILE_LICENSE"/>
</application>
```

For iOS add it as a key with a value in the project specific Info.plist file:

```
<key>MX_MOBILE_LICENSE</key>
<string>Your license key</string>
```

Implementing SDK

1. When you build your UI considering that if you want to show partial screen you need to set size on **scannerControl** that represent the reader device and when you create camera scanner in constructor send **false** as input parameter for **fullScreen**. If you want to use full screen preview don't need to set any size on **scannerControl** and in constructor send **true** as input parameter for **fullScreen** (like in our sample app).

2. Set up the following event handlers to monitor the connection state of the scannerControl and receive information about the read code:

```
// 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 i
public void OnAvailabilityChanged(object sender, ScannerAvailability availability)
{
    ClearResult();

    if (availability == ScannerAvailability.Available)
    {
        ConnectToScannerDevice();
    }
    else if (availability == ScannerAvailability.Unavailable)
    {
        DisplayAlert("Device became unavailable", null, "OK");
    }
}

// The connect method has completed, here you can see whether there was an error with establishing the connection or no
// (args: ScannerExceptions exception, string errorMessage)
public void OnConnectionCompleted(object sender, object[] args)
{
    // If we have valid connection error param will be null,
    // otherwise here is error that inform us about issue that we have while connecting to scanner
    if ((ScannerExceptions)args[0] != ScannerExceptions.NoException)
    {
        // ask for Camera Permission if necessary (android only, for iOS we handle permission from SDK)
        if ((ScannerExceptions)args[0] == ScannerExceptions.CameraPermissionException)
            RequestCameraPermission();
        else
        {
            Debug.WriteLine(args[1].ToString());
            UpdateUIByConnectionState(ScannerConnectionStatus.Disconnected);
        }
    }
}

// This is called when a connection with the scanner has been changed.
// The scanner is usable only in the "Connected" state
public void OnConnectionStateChanged(object sender, ScannerConnectionStatus status)
{
    ClearResult();

    if (status == ScannerConnectionStatus.Connected)
    {
        // We just connected, so now configure the device how we want it
        ConfigureScannerDevice();
    }

    isScanning = false;
```

```
    UpdateUIByConnectionState(status);
}

// This is called after scanning has completed, either by detecting a barcode, canceling the scan by using the on-scree
public void OnReadResultReceived(object sender, List<ScannedResult> results)
{
    ClearResult();

    foreach (ScannedResult item in results)
        scanResults.Add(item);

    isScanning = false;
    btnScan.Text = "START SCANNING";
}
```

3. Create scanner device

Using the MX Reader

Initialize a scanner device object for MX readers using the following factory method:

```
//**************************************************************************************
                // Create an MX-1xxx scanner  (note that no license key in needed)
                //**************************************************************************************
scannerControl.GetMXDevice();
```

The availability of the MX mobile terminal can change when the device turns on or off, or if the USB cable gets connected or disconnected. You can handle those changes using the following method from *AvailabilityChanged* event handler:

```
public void OnAvailabilityChanged(ScannerAvailability args)
```

Using the Camera Reader

You are recommended to use an MX mobile terminal to scan barcodes. However, cmbSDK also supports using the built-in camera of a mobile device. This includes the support of optional external aimers or illumination, and the customization of the live-stream preview's appearance.

To scan barcodes using the built-in camera of a mobile device, initialize the *scanner device* object using the *scannerControl.GetPhoneCameraDevice* method. The camera reader has several options when initialized. The following parameters are required:

- *CameraMode*
- *PreviewOption*
- *FullScreen*
- *RegistrationKey*

The *ScannerCameraMode* parameter is of type *ScannerCameraMode* enum and it accepts one of the values listed in the following table.

These modes provide the following default settings for the scanner:

- The zoom feature is available and a button to control it is visible on the live-stream preview (if displayed).
- The simulated hardware trigger (volume control buttons) is disabled.
- When *StartScanning()* is called, the decoding process is started.

Based on the selected mode, additional illumination options and behaviors are set, also listed in the table.

VALUE	DESCRIPTION	ILLUMINATION	LIVE-STREAM PREVIEW
NoAimer	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 if the mobile device does not have an aiming accessory.	Illumination is available and a button to control it is visible on the live-stream preview.	Displayed
		If commands are sent to the reader for aimer control, they are ignored.	
PassiveAimer	Initializes the reader to use a passive aimer. No live-stream preview is available on the device screen in this mode, since an aiming pattern is projected.	Illumination is not available, and the live-stream preview does not have an illumination button.	Not Displayed
		If commands are sent to the reader for illumination control, they are ignored because it is assumed in this mode that the built-in LED of the mobile device is being used for the aimer.	
ActiveAimer (for iOS only)	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 .	Not Displayed
		Illumination or aimer control commands are accepted .	
FrontCamera	Initializes the reader to use the front camera of the mobile device, if available. Use this configuration with care because most front facing cameras do not have auto focus and illumination, and provide significantly lower resolution images. Illumination is not available in this mode.	The front camera is used.	Displayed
		Illumination is not available and the live-stream preview does not have an illumination button.	
		If commands are sent to the reader for aimer or illumination control, they are ignored.	

The Scanner*PreviewOption* parameter is of type *ScannerPreviewOption* enum, and is used to change the reader's default values or override defaults derived from the selected Scanner*CameraMode*. You can specify the following options:

VALUE	DESCRIPTION
Defaults	Accept all defaults set by the CameraMode.
NoZoomButton	Hides the zoom button on the live-stream preview, preventing the user from adjusting the zoom of the mobile device camera.
NoIlluminationButton	Hides the illumination button on the live-stream preview, preventing the user from toggling the illumination.
HardwareTrigger	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 purpose-built scanner's trigger, and pressing it a second time does not stop the scanning process.
Paused	If using a live-stream preview, the preview is displayed when the *startScanning()* method is called, but the reader does not start decoding until the user presses the on-screen button to start the scanning process.
AlwaysShow	Forces a live-stream preview to be displayed even if an aiming mode is selected (for example CameraMode == PASSIVE_AIMER).
PessimisticCaching	Uses the device camera in higher resolution, changing the default 1280x720 resolution to 1920x1080 on devices that support it, and to the default resolution on devices that do not support it. This can help with scanning small barcodes, but increases the decoding time as there is more data to process in each frame.
HighResolution (for iOS only)	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 .
HighFrameRate	Uses the device's camera in 60 FPS instead of the default 30 FPS to provide a smoother camera preview.
ShowCloseButton	Show close button in partial view.

If the *FullScreen* parameter is set **true**, a full screen preview is used, otherwise partial screen preview is in use.

The *RegistrationKey (optional)* parameter is used to license your SDK with license key that you have

Examples:

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

```
scannerControl.GetPhoneCameraDevice(ScannerCameraMode.NoAimer, ScannerPreviewOption.Defaults, true);
```

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

```
scannerControl.GetPhoneCameraDevice(ScannerCameraMode.NoAimer, ScannerPreviewOption.NoZoomButton | ScannerPreviewOption.Har
```

Requesting Camera Permission for Phone Camera Scanner (for Android only)

From Android 6.0 and above you need to request permission from the user to access the built-in camera of the mobile device.

If the camera cannot be opened due to permission issues, the *OnConnectionCompleted(sender, args)* callback contains a *ScannerException* in the args parameter. You can check for this exception type and request permission.

```
public void OnConnectionCompleted(object sender, object[] args)
{
    // If we have valid connection error param will be null,
    // otherwise here is error that inform us about issue that we have while connecting to scanner
    if ((ScannerExceptions)args[0] != ScannerExceptions.NoException)
    {
        // ask for Camera Permission if necessary (android only, for iOS we handle permission from SDK)
        if ((ScannerExceptions)args[0] == ScannerExceptions.CameraPermissionException)
            RequestCameraPermission();

...
```

If camera permission is granted you can try to connect on scanner device again:

```
private async void RequestCameraPermission()
{
    var result = await Permissions.RequestAsync<Permissions.Camera>();

    // Check result from permission request. If it is allowed by the user, connect to scanner
    if (result == PermissionStatus.Granted)
    {
        scannerControl.Connect();
    }
    else
    {
        if (Permissions.ShouldShowRationale<Permissions.Camera>())
        {
            if (await DisplayAlert(null, "You need to allow access to the Camera", "OK", "Cancel"))
                RequestCameraPermission();
        }
    }
}
```

Connecting to the Device

Before connecting, set the *OnAvailabilityChanged*, *ConnectionStateChanged* and *ConnectionCompleted* event handlers.

```
public void OnAvailabilityChanged(object sender, ScannerAvailability availability)
public void OnConnectionStateChanged(object sender, ScannerConnectionStatus status)
public void OnConnectionCompleted(object sender, object[] args)
```

Invoke the connect method after initializing the *ScannerDevice*.

```
// Before the scanner can be configured or used, a connection needs to be established
private void ConnectToScannerDevice()
{
```

```
    scannerControl.Connect();
}
```

Event handlers that we set before will be called with new *ScannerDevice* status information.

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 are 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 (for iOS only)

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: **PreviewOption.PessimisticCaching**.

Enabling Symbologies

CmbSDK does not enable any *symbologies* by default for barcode reading with the built-in camera of the mobile device. You must enable all barcode symbologies your application needs to scan to achieve optimal scanning performance.

Individual symbologies can be enabled using the following method of the *ScannerControl* class:

```
public void SetSymbologyEnabled(Symbology symbology, bool enable)
```

All symbologies used for the symbology parameter in this method can be found in **Symbology** enum.

Examples

```
// Explicitly enable the symbologies we need
scannerControl.SetSymbologyEnabled(Symbology.Datamatrix, true);
scannerControl.SetSymbologyEnabled(Symbology.C128, true);
```

You can also use the same method to disable symbologies:

```
// Explicitly disable symbologies we know we don't need
scannerControl.SetSymbologyEnabled(Symbology.Codabar, false);
```

You can implement SymbologyEnabled event handler to check the result of the symbology change:

```
// SymbologyEnabled listener (args: Symbology symbology, bool isEnabled, string error)
public void OnSymbologyEnabled(object sender, object[] args)
{
    if ((string)args[2] != null)
        Debug.WriteLine("Failed to enable/disable " + args[0].ToString());
    else
        Debug.WriteLine(((Symbology)args[0]).ToString() + ((bool)args[1] ? " enabled" : " disabled"));
}
```

Illumination Control

If your scanner device is equipped with illumination lights, you can control them using this DMCC:

```
scannerControl.SendCommand("SET LIGHT.INTERNAL-ENABLE ON");
```

You can control lights while scanning preview is active, or if you send this DMCC before starting the scanner you will set the initial value for lights.

Not all devices and device modes support illumination control.

Camera Zoom Settings

If the built-in camera of a mobile device is used as the reader device, you can configure zoom levels and how they are used. There are three zoom levels:

- normal: not zoomed (100%)
- level 1 zoom (150% on Android by default)
- level 2 zoom (300% on Android by default)

The *SET CAMERA.ZOOM-PERCENT [100-MAX] [100-MAX]* command is for configuring how far the two levels zoom in percentage. 100 is not zoomed and MAX (goes up to 1000) zooms as far as the device is capable of. The first argument is used for setting level 1 zoom, and the second for level 2 zoom.

You can check the current zoom setting with the *GET CAMERA.ZOOM-PERCENT* command, which returns two values: level 1 and level 2 zoom.

Example

```
scannerControl.SendCommand("SET CAMERA.ZOOM-PERCENT 250 500");
```

Note: The camera needs to be started within cmbSDK 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 only go up to 600, the *GET CAMERA.ZOOM-PERCENT* command returns 1000 as long as camera is not opened, but it returns 600 afterwards.

GET/SET CAMERA.ZOOM 0-2 is another command that sets the zoom level or returns the actual setting. Possible values for the SET command are:

- 0 - normal (not zoomed)
- 1 - level 1 zoom
- 2 - level 2 zoom

You can call this command before or even during scanning, and the zoom goes up to the configured level. If scanning is finished, the value is reset to normal behavior (0).

Example

```
scannerControl.SendCommand("SET CAMERA.ZOOM 2");
```

Camera Overlay Customization

When using the mobile device's camera, cmbSDK allows you to see the camera preview inside a preview container or in full screen. This preview also contains a customizable overlay. The cmbSDK camera overlay features buttons for zooming, flashing and closing the scanner, and a progress bar indicating the scan timeout.

To use the legacy camera overlay originally used in cmbSDK v2.0.x and ManateeWorks SDK, use this before initializing the ScannerDevice:

```
private void CreateScannerDevice()
{

    if (param_deviceClass == ScannerDevice.PhoneCamera)
    {
        scannerControl.SetOverlay(ScannerOverlay.LEGACY);
```

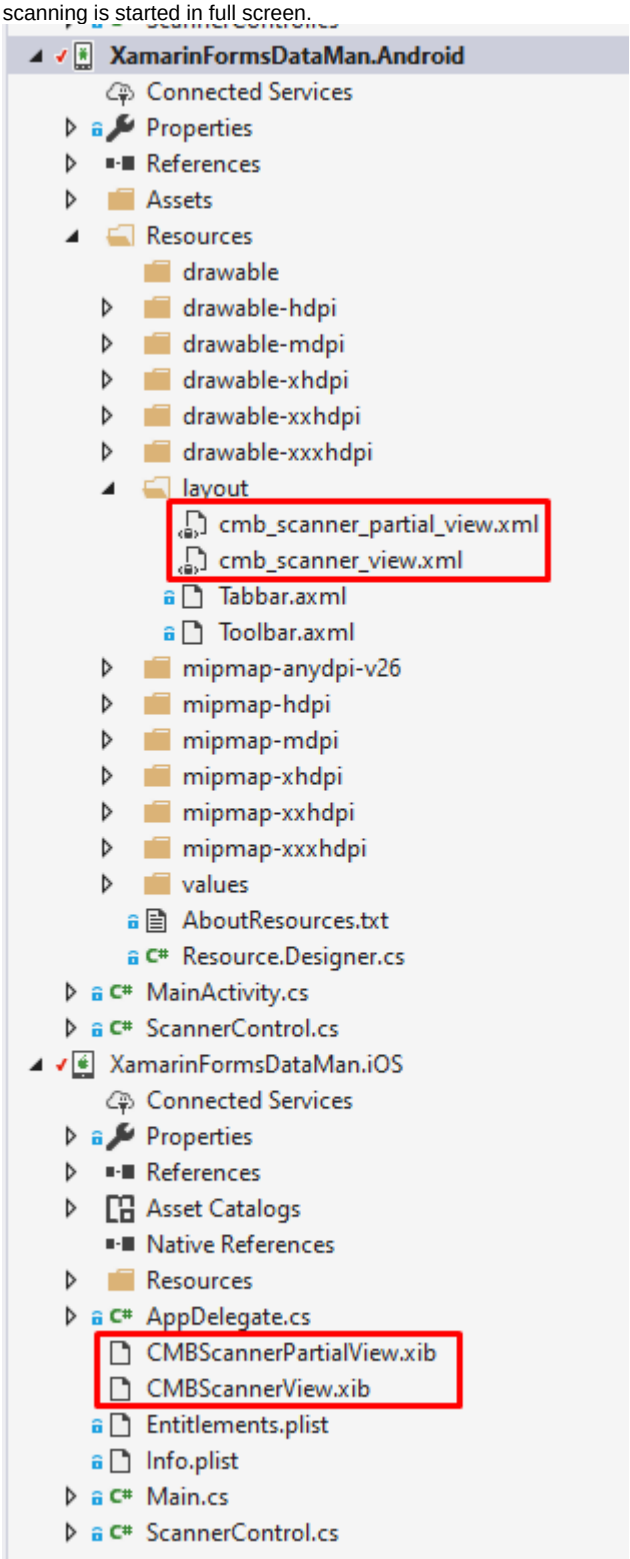
```
        scannerControl.GetPhoneCameraDevice(ScannerCameraMode.NoAimer, ScannerPreviewOption.Defaults, true);
    }
```

The customization of the legacy camera overlay is limited, so it is recommended to use the cmbSDK overlay.

When using the cmbSDK overlay:

1. Copy the layout files from the Resources/Android/layout directory into your Android platform specific project and modify them. Use **cmb_scanner_partial_view.xml** if scanning is started inside a container (partial view), and use **cmb_scanner_view.xml** if scanning is started in full screen. Same for iOS, copy the xib files from the Resources/iOS directory into your iOS platform specific project and modify them. Use **CMBScannerPartialView.xib** if scanning is started inside a container (partial view), and use **CMBScannerView.xib** if

scanning is started in full screen.



2. Modify the layout and xib according to your needs. For example, you can change the sizes, positions or color of the views, remove views and add your own views, like an overlay image.

CmbSDK accesses the views it uses (zoom, flash, close buttons, the view used for drawing lines on the corners, and the progress bar) with the tag attribute. Do not change the tag attribute, otherwise cmbSDK cannot recognize the views and continues to function as if they are removed.

Both the cmbSDK and the legacy overlay allow you to change the images used on the zoom and flash buttons if your images have the same name as the names cmbSDK uses. You can find the images and names used in cmbSDK in the Resources/**drawable-mdpi** and **drawable-hdpi** directories for Android and Resources/**MWBScannerImages.xcassets** for iOS.

Both the cmbSDK and the legacy overlay allow you to change the color and width of the rectangle that is displayed when a barcode is detected. These changes need to be done inside ConfigureScannerDevice() method (after we have valid connection to scanner device).

Example:

```
private void ConfigureScannerDevice()
{
   scannerControl.SetLegacyOverlayLocationLine(255, System.Drawing.Color.Yellow, 6.0f, true);

...
```

Advanced Configuration using DataMan Control Commands

Cognex scanning devices implement DataMan Control Commands (DMCC) for configuring and controlling the device. Every feature of the device can be controlled using this text-based language. The API provides a method for sending DMC commands to the device. Commands exist both for setting and querying configuration properties. DMC commands are same for all platforms and frameworks.

The Appendix includes the complete DMCC reference for the camera reader.

> The DMCCs for MX mobile terminals and other supported devices can be found in their respective manuals available through Setup Tool.

The following examples show different DMCC sent to the device for more advanced configuration.

Examples

```
//Change the scan direction to omnidirectional
scannerControl.SendCommand("SET DECODER.1D-SYMBOLORIENTATION 0");
//Change live-stream preview's scanning timeout to 10 seconds
scannerControl.SendCommand("SET DECODER.MAX-SCAN-TIMEOUT 10");
```

You can also set ResponseReceived event handler to receive response from the send command:

```
// ResponseReceived listener after we send DMCC command (args: string payload, string error, string dmcc)
public void OnResponseReceived(object sender, object[] args)
{
    if ((string)args[1] != null)
        Debug.WriteLine("Failed to execute DMCC");
    else
        Debug.WriteLine("Response for " + (string)args[2] + ": " + (string)args[0]);
}
```

Resetting the Configuration

> NOTE: This section includes resetting to CmbSDK defaults and does not include instruction on resetting to factory defaults.

CmbSDK includes a method for resetting the device to its default settings. In case of an MX mobile terminal, the default setting are the saved configurations. In case of a built-in camera, the default settings are the defaults identified in the Appendix, where no symbologies are enabled.

To reset the device send this DMCC:

```
scannerControl.SendCommand("CONFIG.DEFAULT");
```

When using an MX mobile terminal, there are three states that we can distinguish:

- Factory defaults
- Saved configuration: when there were different configurations set on the device and CONFIG.SAVE DMCC was called.
- Session configuration: when you make changes on the saved configuration, the changes are valid until the MX Mobile Terminal is rebooted. If it is rebooted, it has the saved configuration state.

Scanning Barcodes

With a properly configured reader, you are ready to scan barcodes. This is simply accomplished by calling the **StartScanning()** method from your *scannerControl*. What happens next is based on the type of *ScannerDevice* 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 ScannedResult list in your ReadResultReceived event:

```
// This is called after scanning has completed, either by detecting a barcode, canceling the scan by using the on-screen bu
public void OnReadResultReceived(object sender, List<ScannedResult> results)
```

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

```
public void OnReadResultReceived(object sender, List<ScannedResult> results)
{
    if (results.Count > 0)
    {
        ScannedResult readResult = results[0];

        lblCode.Text = readResult.ResultCode;
    }
}
```

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 in the *ConnectionStateChanged* event handler.

Note: The **OnAvailabilityChanged** 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 *scanner device* object before trying to call the **Connect** 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. For iOS we are doing that in ScannerControl renderer while for Android need to be done manually when page is resumed:

```
// When an page is disappeared, the connection to the scanning device needs
// to be closed; thus when we are resumed (Appear this page again) we
// have to restore the connection (assuming we had one).
// This is used for android only.
// For iOS we use Observer that is created in ScannerControl class in iOS platform specific project
//------------------------------------------------------------------------
if (Device.RuntimePlatform == Device.Android && await Permissions.CheckStatusAsync<Permissions.Camera>() == PermissionStatu
{
    ConnectToScannerDevice();
}
```

Migration from mwSDK to cmbSDK

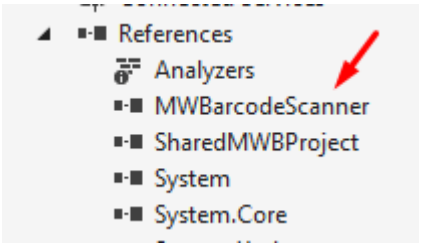
Difference between mwSDK and cmbSDK

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 you need to remove reference that refer to old **MWBarcodeScanner.dll** file in your platform specific project.



Add cmbSDK

Next copy new **XamarinDataManLibrary.dll** file that contain cmbSDK in your platform specific project directory, and add new reference to that dll.

Please navigate to this url to check step by step how to integrate cmbSDK inside your Xamarin 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 **ReaderDevice** object and we use API methods from **BarcodeScanner** object to initialize decoder before starting the scanner process: set active codes, set scanning rect, set decoder level, register sdk, etc. When we use cmbSDK all this things are done in code behind with default values when we create ReaderDevice object. Here some of the settings can be set in constructor as input parameter and another one 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 callbacks that will handle response from connection state changed, availability, result received, etc.

2. **Start scanning process**
   - With mwSDK after we initialize decoder we are ready to start the scanning. There are two different methods for starting the scanner.One for partial view (where we can set size as input parameter) and one for full screen. Also we can choose if we want this methods to return object result or we will expect result in a callback function.
   Using cmbSDK there is only one method to start the scanning process and comes from ReaderDevice object (readerDevice.startScanning()). We can't start scanning process if we don't have valid connection to reader device. Here if we want to use full screen mode when we create reader device in constructor will send **null** as input parameter for previewContainer, or if we want to use partial view we must to create that container in our layout and send as input parameter. Result from scanning process will be received in onReadResultReceived callback function.

3. **Result received**
   - If we have successful read or we stop the scanning process and have no read, result object will be received in onReadResultReceived. In cmbSDK result object is more extended than in mwSDK. From that object we can read our barcode result, symbology, image from last frame, SVG result, etc.