

## How to guides (v2.7.x)

### Target Decoding

Target decoding simulates the behavior of a laser barcode scanner. Instead of looking at the entire field of view, however, the reader attempts to decode barcodes that lie only under the laser aiming dot. This prevents the operator from accidentally reading other nearby barcodes.

#### Supported devices:

- MX mobile terminals
- MX barcode reader
- Camera

#### Configuration DMCCs:

- **(GET|SET) DECODER.TARGET-DECODING [ON|OFF]**

Enable or disable target decoding (or retrieve it's state).

Default value: OFF

#### Example

```
readerDevice.getDataManSystem().sendCommand("SET DECODER.TARGET-DECODING ON");
```

- [Android](#)
- [Swift](#)
- [Objective-C](#)

```
readerDevice.getDataManSystem().sendCommand("SET DECODER.TARGET-DECODING ON");
```

```
self.readerDevice.dataManSystem().sendCommand("SET DECODER.TARGET-DECODING ON")
```

```
[self.readerDevice.dataManSystem sendCommand:@"SET DECODER.TARGET-DECODING ON"];
```

- **(GET|SET) DECODER.CENTERING-WINDOW [centerX centerY sizeX sizeY]**

Set (or get) the size of the centering window. The numbers are in percent of the image.

- centerX, centerY: location of centering window as percentage of the sensor size (50 - 50 means middle on both axis)
- sizeX and sizeY: size of centering window also as percentage

If a barcode has overlap with the centering window the that barcode is decoded. So the centering window does not necessarily need to contain the whole barcode .

Default value: 50 50 10 10

#### Example

- [Android](#)
- [Swift](#)
- [Objective-C](#)

```
readerDevice.getDataManSystem().sendCommand("SET DECODER.CENTERING-WINDOW 50 50 5 5");
```

```
self.readerDevice.dataManSystem().sendCommand("SET DECODER.CENTERING-WINDOW 50 50 5 5")
```

```
[self.readerDevice.dataManSystem sendCommand:@"SET DECODER.CENTERING-WINDOW 50 50 5 5"];
```

- **(GET|SET) DECODER.DISPLAY-TARGET [ON|OFF]**

Set (or get) if the centering window is added to the SVG result.

Default value: ON

#### Example

- [Android](#)
- [Swift](#)
- [Objective-C](#)

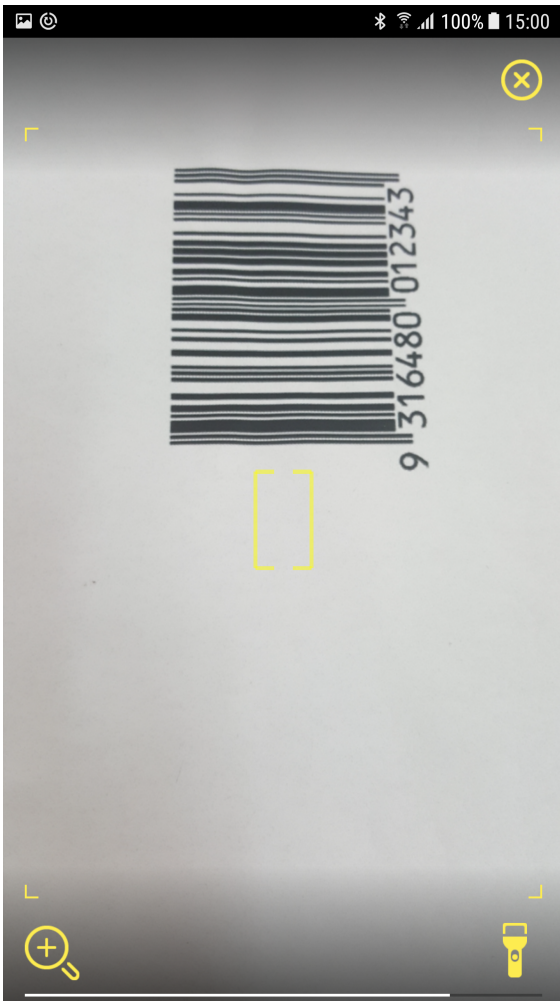
```
readerDevice.getDataManSystem().sendCommand("SET DECODER.DISPLAY-TARGET ON");
```

```
self.readerDevice.dataManSystem().sendCommand("SET DECODER.DISPLAY-TARGET ON")
```

```
[self.readerDevice.dataManSystem sendCommand:@"SET DECODER.DISPLAY-TARGET ON"];
```

#### Remark

Here is example of camera's scanning preview when target decoding is enabled in the camera AP (yellow, smaller rectangle is the target decoding window):



Target decoding window (rectangle) color and weight of the lines can be changed with:

- [Android](#)
- [Swift](#)
- [Objective-C](#)

```
MWOverlay.targetRectLineColor = Color.RED;  
MWOverlay.targetRectLineWidth = 2;
```

```
MWOverlay.setTargetRectLineUIColor(UIColor.red)  
MWOverlay.setTargetRectLineWidth(2)
```

```
[MWOverlay setTargetRectLineUIColor:[UIColor redColor]];  
[MWOverlay setTargetRectLineWidth:2];
```

## Multicode

If you want to scan more barcodes at once and in one scanning session you can use **Multicode reading**.

## Supported devices:

- MX mobile terminals
- MX barcode reader
- Camera

## Configuration DMCCs:

- **(GET|SET) MULTICODE.NUM-CODES [1-255]**

Set number of codes that the decoder should find for a successful read result.

Default value: 1

## Example

- [Android](#)
- [Swift](#)
- [Objective-C](#)

```
readerDevice.getDataManSystem().sendCommand("SET MULTICODE.NUM-CODES 3");
```

```
self.readerDevice.dataManSystem().sendCommand("SET MULTICODE.NUM-CODES 3")
```

```
[self.readerDevice.dataManSystem sendCommand:@"SET MULTICODE.NUM-CODES 3"];
```

- **(GET|SET) MULTICODE.PARTIAL-RESULTS [ON|OFF]**

Define how the reader interprets the number of codes to find.

- ON = reader will return a successful read if at least 1 barcode was found
- OFF = reader will return a successful read only if number of codes found equals to the value that is set in MULTICODE.NUM-CODES.

Default value: OFF

## Example

- [Android](#)
- [Swift](#)
- [Objective-C](#)

```
readerDevice.getDataManSystem().sendCommand("SET MULTICODE.PARTIAL-RESULTS ON");
```

```
self.readerDevice.dataManSystem().sendCommand("SET MULTICODE.PARTIAL-RESULTS ON")
```

```
[self.readerDevice.dataManSystem sendCommand:@"SET MULTICODE.PARTIAL-RESULTS ON"];
```

- **(GET|SET) DECODER.REREAD-NOT-LAST-N [0-100]**

Do not read code if this code was read within the last N reads.

Default value: 0 (no restriction)

#### Example

- [Android](#)
- [Swift](#)
- [Objective-C](#)

```
readerDevice.getDataManSystem().sendCommand("SET DECODER.REREAD-NOT-LAST-N 1");
```

```
self.readerDevice.dataManSystem().sendCommand("SET DECODER.REREAD-NOT-LAST-N 1")
```

```
[self.readerDevice.dataManSystem sendCommand:@"SET DECODER.REREAD-NOT-LAST-N 1"];
```

- **(GET|SET) MULTICODE.MAX-NUM-CODES [0-5] [0-255]**

Define expected maximum number of codes to find for each symbology group:

0. Sets the same maximum number of codes for each groups. In this case the second parameter cannot be 0.

1. DataMatrix
2. QR Code/MaxiCode/AztecCode
3. Linear/ Postal/ Stacked
4. VeriCode
5. DotCode

No expected value for any single symbology can exceed parameter MULTICODE.NUM-CODES, the total number of codes to find. 1st param is symbology group, second one is max number of codes.

Default value: 1 for all group

#### Example

- [Android](#)

- [Swift](#)
- [Objective-C](#)

```
readerDevice.getDataManSystem().sendCommand("SET MULTICODE.MAX-NUM-CODES 1 3");
```

```
self.readerDevice.dataManSystem().sendCommand("SET MULTICODE.MAX-NUM-CODES 1 3")
```

```
[self.readerDevice.dataManSystem sendCommand:@"SET MULTICODE.MAX-NUM-CODES 1 3"];
```

## Remark

If multicode reading is enabled, you get in the `onReadResultReceived` callback function the list of all results by using `getSubResults()` method of `ReadResults` object. If multicode reading is disabled this method returns null.

## Parsers

Parser help us to extract decoded results into a structured format (JSON, Key-Value) for search, sort, and validation. There are six types of parser:

1. AAMVA
2. GS1
3. HIBC
4. ISBT128
5. IUD
6. SCM

Also there is option to use AUTO parser type in `cmbSDK` and it tries to find which one fits the best.

### Supported devices:

- MX mobile terminals
- MX barcode reader
- Camera

Use `setParser()` method from `ReaderDevice` object to set parser type that you want to use, or `getParser()` to get selected type. Set parser type after valid connection to reader device. **None by default.**

### Example

- [Android](#)
- [Swift](#)
- [Objective-C](#)

```
readerDevice.setParser(ResultParser.AAMVA);
```

```
self.readerDevice.parser = CMBResultParserAAMVA
```

```
self.readerDevice.parser = CMBResultParserAAMVA;
```

You can get structured format from received result in `onReadResultReceived` callback function of `ReadResult` after you enable the required parser type:

- [Android](#)
- [Swift](#)
- [Objective-C](#)

```
//Returns parsed text in json format from the result
results.getResultAt(0).getParsedJSON();
//Returns parsed text from the result
results.getResultAt(0).getParsedText();
```

```
let results:[CMBReadResult] = readResults?.readResults as? [CMBReadResult] ?? []
//Returns parsed text in json format from the result
let parsedJSON = results.first?.parsedJSON
//Returns parsed text from the result
let parsedText = results.first?.parsedText
```

```
NSArray<CMBReadResult*>* results = readResults.readResults;
//Returns parsed text in json format from the result
NSString* parsedJSON = [[results firstObject] parsedJSON];
//Returns parsed text from the result
NSString* parsedText = [[results firstObject] parsedText];
```

## Region of Interest (ROI)

You can use this feature to specify the exact region of the image that will be used to locate and decode a symbol. This can be helpful if two barcodes are in close proximity to each other and you need to ensure you only read the code along the target area. Reducing the region of interest does not necessarily reduce the decoding time or increase the image transfer rate.

### Supported devices:

- MX mobile terminals
- MX barcode reader
- Camera

### Configuration DMCCs:

#### (GET|SET) DECODER.ROI-PERCENT [X W Y H]

- X and Y represent the starting point of the ROI in each axis
- W and H represent the ROI width and height starting from the starting point, with values in **percentages**.

The range of 0-100 is checked, as well as minimum values of 5 for width and height, and the constraints implied between args, such as the total sum for a given axis to be no more than 100.

### Example

- [Android](#)
- [Swift](#)
- [Objective-C](#)

```
readerDevice.getDataManSystem().sendCommand("SET DECODER.ROI-PERCENT 10 80 10 80");
```

```
self.readerDevice.dataManSystem().sendCommand("SET DECODER.ROI-PERCENT 10 80 10 80")
```

```
[self.readerDevice.dataManSystem sendCommand:@"SET DECODER.ROI-PERCENT 10 80 10 80"];
```

## Scan barcodes from an image

The SDK can decode barcodes from image files (camera API only). Supported image formats include BMP, PNG, and JPG. The image can not be any larger than 5000 x 5000 pixels.

### Supported devices:

- Camera API

To scan an image, your application must first read the image into memory, then use the **IMAGE.LOAD** DMCC command to decode it. The `readerDevice` object must be initialized and configured before sending the **IMAGE.LOAD** command. Below is an example of using the `sendCommand` method to initiate the **IMAGE.LOAD** function.

### Example

- [Android](#)
- [Swift](#)
- [Objective-C](#)

```
readerDevice.getDataManSystem().sendCommand(String.format("IMAGE.LOAD %d", imageData.length), imageData, 500, false, (dataM
    if (response.getError() != null) {
        // IMAGE.LOAD DMCC finished with error, handle the error here
    }
});
```

```
self.readerDevice.dataManSystem()?.sendCommand("IMAGE.LOAD \(imageData.count)", with: imageData, timeout: 100, expectBinary
    if (response?.status != DMCC_STATUS_NO_ERROR) {
        // IMAGE.LOAD DMCC finished with error, handle the error here
    }
})
```

*Note: imageData is the Data of the image.*

```
[self.readerDevice.dataManSystem sendCommand:[NSString stringWithFormat:@"IMAGE.LOAD %lu", [imageData length]] withData:ima
    if (response.status != DMCC_STATUS_NO_ERROR) {
        // IMAGE.LOAD DMCC finished with error, handle the error here
    }
}];
```

*Note: imageData is the NSData of the image.*



Just like with live scanning, when the **IMAGE.LOAD** command completes, results are received in a read result array in your *ReaderDevice*'s delegate using the following method:

- [Android](#)
- [Swift](#)
- [Objective-C](#)

```
void onReadResultReceived(ReaderDevice reader, ReadResults results)
```

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

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

Below you will find an example for selecting and scanning an image from your device's photo library

- [Android](#)
- [Swift](#)
- [Objective-C](#)

```
private void pickImageFromGallery(){
    Intent pickPhoto = new Intent(Intent.ACTION_PICK, android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
    startActivityForResult(pickPhoto, PICK_IMAGE);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {

    if(requestCode == PICK_IMAGE && resultCode == RESULT_OK && data != null && data.getData() != null
        && readerDevice != null && readerDevice.getConnectionState() == ConnectionState.Connected)
    {
        try (InputStream imageStream = getContentResolver().openInputStream(data.getData());
            ByteArrayOutputStream buffer = new ByteArrayOutputStream()) {

            int nRead;
            byte[] bytes = new byte[1024];
            while ((nRead = imageStream.read(bytes, 0, bytes.length)) != -1) {
                buffer.write(bytes, 0, nRead);
            }

            buffer.flush();
            final byte[] byteArray = buffer.toByteArray();

            readerDevice.getDataManSystem().sendCommand(String.format("IMAGE.LOAD %d", byteArray.length), byteArray,
                500, false, (dataManSystem, response) -> {
                    if (response.getError() != null) {
                        Toast.makeText(MainActivity.this, "ERROR [" + response.getError().t
                            response.getError().getLocalizedMessage(), Toast.LENGTH_LONG).show();
                    }
                });

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    super.onActivityResult(requestCode, resultCode, data);
}
```

```

func showImagePicker() {
    let imagePickerController = UIImagePickerController()
    imagePickerController.allowsEditing = false
    imagePickerController.delegate = self
    imagePickerController.sourceType = .photoLibrary

    self.present(imagePickerController, animated: true, completion: nil)
}

// MARK: UIImagePickerControllerDelegate methods

func imagePickerController(_ picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey : Any]) {
    picker.dismiss(animated: true, completion: nil)

    let imagePath = info[UIImagePickerController.InfoKey.imageURL] as! NSURL
    guard let imageData = try? Data.init(contentsOf: imagePath.absoluteURL!) else { return }

    self.readerDevice.dataManSystem()?.sendCommand("IMAGE.LOAD \(imageData.count)", with: imageData, timeout: 100, expectBi
        if response?.status != DMCC_STATUS_NO_ERROR {
            // IMAGE.LOAD DMCC finished with error, handle the error here
        }
    })
}
}

```

```

- (void)showImagePicker {
    UIImagePickerController *imagePickerController = [[UIImagePickerController alloc] init];
    imagePickerController.allowsEditing = NO;
    imagePickerController.delegate = self;
    [imagePickerController setSourceType:UIImagePickerControllerSourceTypePhotoLibrary];

    [self presentViewController:imagePickerController animated:YES completion:nil];
}

#pragma mark - UIImagePickerControllerDelegate

- (void)imagePickerController:(UIImagePickerController *)picker didFinishPickingMediaWithInfo:(NSDictionary *)info{
    [self dismissViewControllerAnimated:YES completion:nil];

    NSURL *imagePath = [info valueForKey:UIImagePickerControllerImageURL];
    NSData *imageData = [NSData dataWithContentsOfURL:imagePath];

    if (imageData) {
        [self.readerDevice.dataManSystem sendCommand:[NSString stringWithFormat:@"IMAGE.LOAD %lu", [imageData length]] with
            if (response.status != DMCC_STATUS_NO_ERROR) {
                // IMAGE.LOAD DMCC finished with error, handle the error here
            }
        }];
    }
}
}

```

## 1D Barcodes Verification

Due to lack of error correction and even checksum values in some one-dimensional barcodes, many are prone to misreads and short-reads (where only a portion of the barcode is decoded). Factors such as poor print quality, inconsistent lighting, blurry images, extreme skew, and others variables can all contribute.

One of the ways the SDK mitigates misreads and short-reads is to use a feature called location verification. When enabled, the SDK will work harder to verify the edges and full location of a one-dimensional barcode; thus greatly minimizing read errors. Be aware that location verification can reduce scanning performance, especially when more than a few symbologies are enabled or when used on value line devices with slower CPUs.

Location verification can be used for the following barcode types (those in bold have this feature enabled by default): **Code 11**, **Code 25**, **Codabar**, Code 93, Code 39, **Code 128**, Telepen, **MSI Plessey**, UPC/EAN.

When combating read errors, consider using location verification in combination with other features of the SDK such as setting minimum read lengths and adjusting the decoder's effort level (see the DMCC command DECODER.EFFORT).

## Supported devices:

- Camera API

## Configuration API:

- [Android](#)
- [Swift](#)
- [Objective-C](#)
- **GET:** `MWBgetParam([1D_CODE_MASK], MWB_PAR_ID_VERIFY_LOCATION)`
- **SET:** `MWBsetParam([1D_CODE_MASK], MWB_PAR_ID_VERIFY_LOCATION, [STATUS])`
  - 1D\_CODE\_MASK: barcode type (check OS specific cmbSDK API reference for further details)
  - STATUS can be:
    - `MWB_PAR_VALUE_VERIFY_LOCATION_ON` to enable verification
    - `MWB_PAR_VALUE_VERIFY_LOCATION_OFF` to disable verification
- **GET:** `MWB_getParam([1D_CODE_MASK], MWB_PAR_ID_VERIFY_LOCATION)`
- **SET:** `MWB_setParam([1D_CODE_MASK], MWB_PAR_ID_VERIFY_LOCATION, [STATUS])`
  - 1D\_CODE\_MASK: barcode type (check OS specific cmbSDK API reference for further details)
  - STATUS can be:
    - `MWB_PAR_VALUE_VERIFY_LOCATION_ON` to enable verification
    - `MWB_PAR_VALUE_VERIFY_LOCATION_OFF` to disable verification
- **GET:** `MWB_getParam([1D_CODE_MASK], MWB_PAR_ID_VERIFY_LOCATION)`
- **SET:** `MWB_setParam([1D_CODE_MASK], MWB_PAR_ID_VERIFY_LOCATION, [STATUS])`
  - 1D\_CODE\_MASK: barcode type (check OS specific cmbSDK API reference for further details)
  - STATUS can be:
    - `MWB_PAR_VALUE_VERIFY_LOCATION_ON` to enable verification
    - `MWB_PAR_VALUE_VERIFY_LOCATION_OFF` to disable verification

## Example

- [Android](#)
- [Swift](#)
- [Objective-C](#)
- 

```
BarcodeScanner.MWBsetParam(BarcodeScanner.MWB_CODE_MASK_39, BarcodeScanner.MWB_PAR_ID_VERIFY_LOCATION, BarcodeScanner.MWB_P
```

```
MWB_setParam(MWB_CODE_MASK_39, UInt32(MWB_PAR_ID_VERIFY_LOCATION), UInt32(MWB_PAR_VALUE_VERIFY_LOCATION_ON))
```

```
MWB_setParam(MWB_CODE_MASK_39, MWB_PAR_ID_VERIFY_LOCATION, MWB_PAR_VALUE_VERIFY_LOCATION_ON);
```

## Partial screen preview

By default when you create reader device using camera API, full screen preview is in use. If you want to use partial screen preview you need to create **ViewGroup** container in your layout for android or **View** container in your main storyboard for iOS and later send that container as parameter in reader device constructor, otherwise if that parameter is not set or if **null** is set, full screen preview will be used.

When partial screen preview is in use, frames from the camera are filled inside preview container, and if preview container have aspect ratio different than camera aspect ratio we are performing cropping to avoid distorsion.

### Supported devices:

- Camera API

### Example

First create container in your layout / storyboard.

- [Android](#)
- [iOS](#)

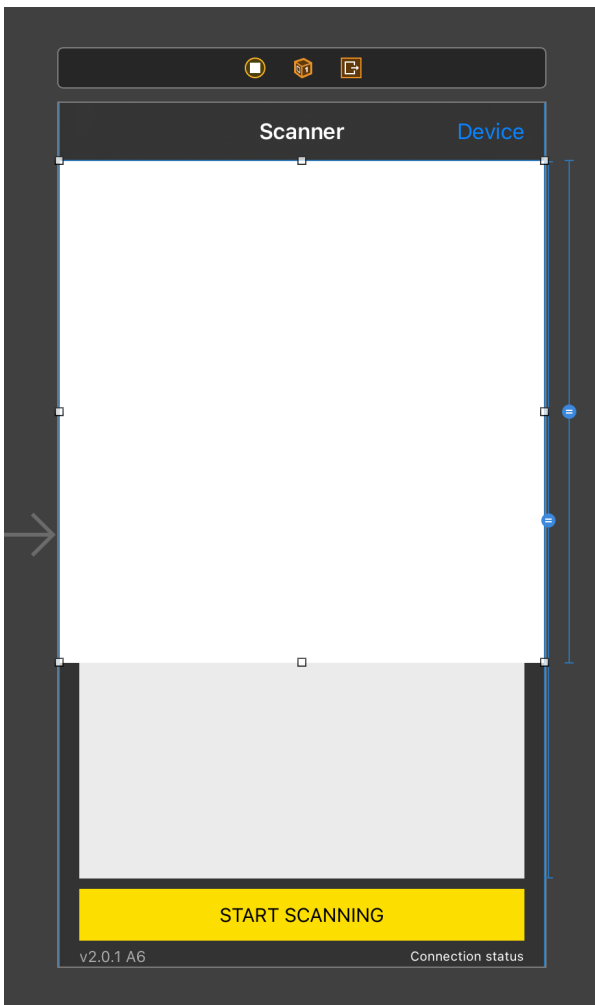
```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#333333"
    tools:context="cognex.com.cmbcamerademo.ScannerActivity">

    <RelativeLayout
        android:id="@+id/rlPreviewContainer"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        app:layout_constraintBottom_toTopOf="@id/listResult"
        app:layout_constraintStart_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    ....

```

Add a UIView in your storyboard. This view will be used as a container for the camera preview. Add constraints to set it's position and size to represent how you would like the camera preview to be displayed in your app.



Later when you create reader device send that container as parameter in reader device constructor

- [Android](#)
- [Swift](#)
- [Objective-C](#)

```
readerDevice = ReaderDevice.getPhoneCameraDevice(this,  
    param_cameraMode, PreviewOption.DEFAULTS, (ViewGroup) findViewById(R.id.rlPreviewContainer));
```

Connect the newly added UIView to an outlet in your ViewController

```
@IBOutlet weak var previewContainer: UIView!
```

Then create the readerDevice using the previewContainer as a parameter in the constructor

```
let readerDevice = CMBReaderDevice.readerOfDeviceCamera(with: self.cameraMode, previewOptions:CDMPreviewOption.init(rawValu
```

Connect the newly added UIView to an outlet in your ViewController

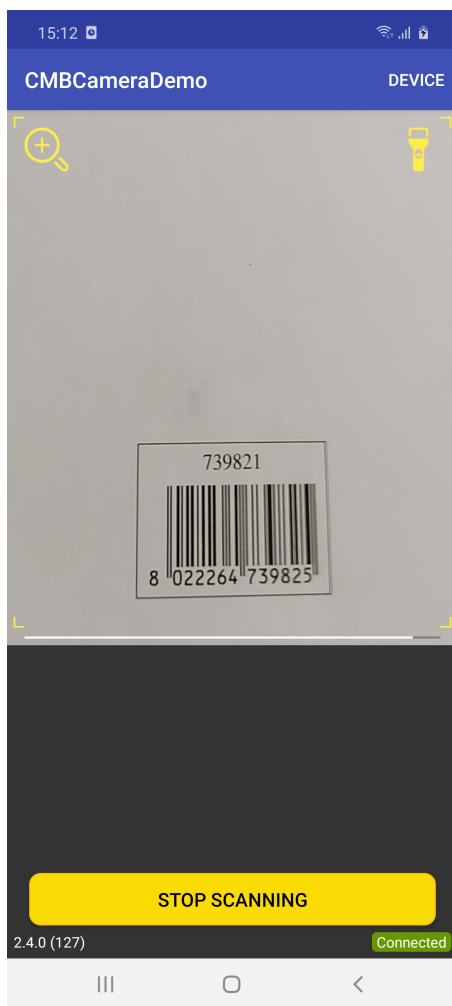
```
@property (weak, nonatomic) IBOutlet UIView *previewContainer;
```

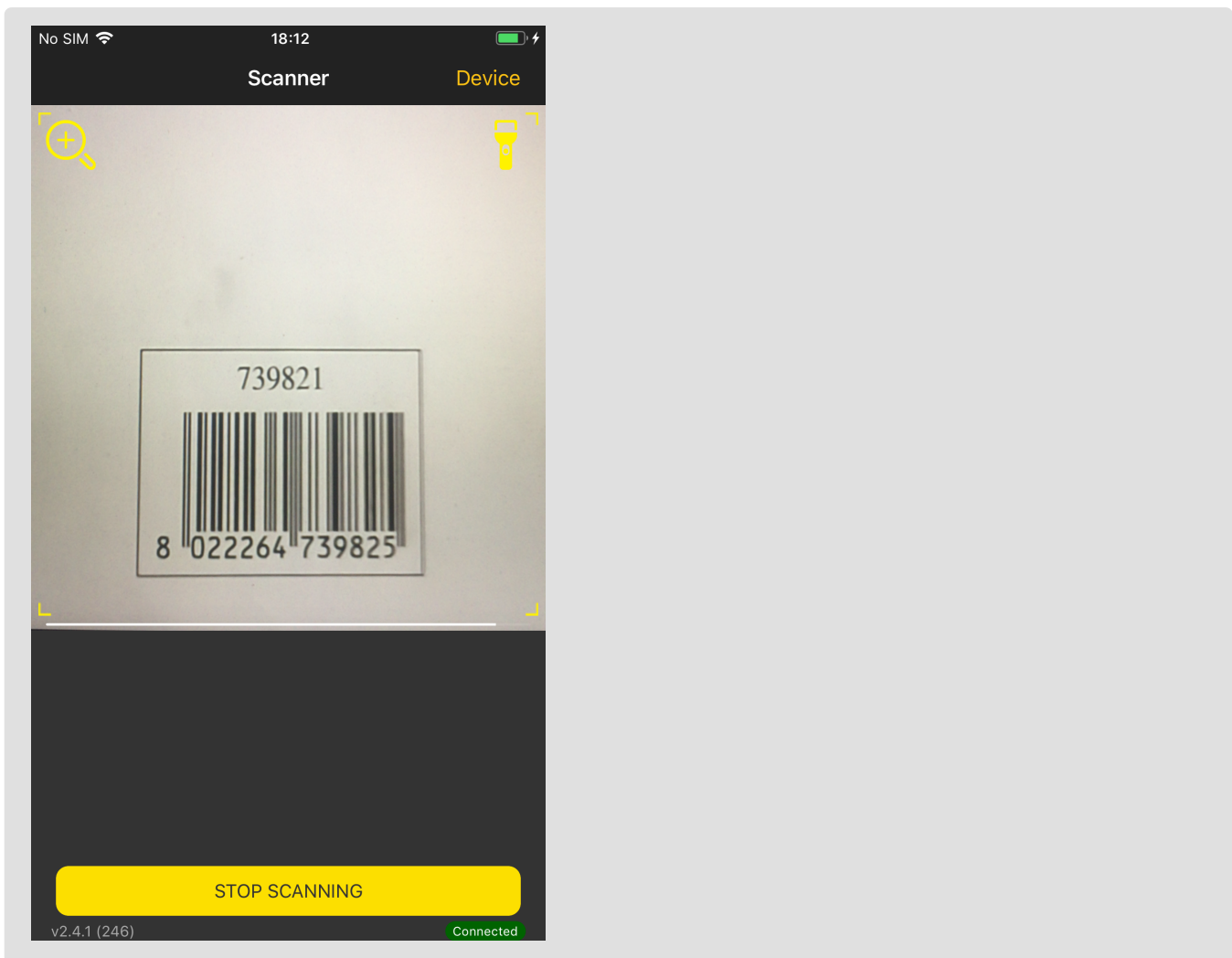
Then create the readerDevice using the previewContainer as a parameter in the constructor

```
CMBReaderDevice *readerDevice = [CMBReaderDevice readerOfDeviceCameraWithCameraMode:cameraMode previewOptions:kCDMPReviewOp
```

Here is how it looks

- [Android](#)
- [iOS](#)





## Handle focusing issues in our legacy SDK and iPhones with triple cameras setup

Some of latest iPhone models with multiple built-in cameras have increased "minimum focus distance" (usually 200mm) on their default camera. That means if you are using the default camera and holding the device too close to the object the image will be blurry and scanning is likely to fail.

The solution is to let the system use multiple cameras and select the appropriate one on the fly. This way the system will switch to ultra wide camera if it detects that the object is too close for the default camera. The ultra wide camera has smaller "minimum focus distance" and will produce sharper images. In your code instead of retrieving the default camera **you should retrieve a virtual camera consisting of multiple cameras.**

```
// let device = AVCaptureDevice.default(for: .video)

let discoverySession = AVCaptureDeviceDiscoverySession(deviceTypes: [.builtInTripleCamera],
mediaType: .video,
position: .back)
let device = discoverySession.devices.first
```

Unfortunately the story doesn't end here. **Two problems remained:**

- *builtInTripleCamera* virtual camera is available only for devices with 3 cameras built-in. **For complete compatibility** you have to use some logic to retrieve the appropriate *AVCaptureDevice*. For this you might use this code snippet:

```

// sortedDeviceTypes parameter should be sorted by priority (1st is the most desired device type)
- (AVCaptureDevice *) selectAvailableDeviceFromSortedTypes:(NSArray<AVCaptureDeviceType> *)sortedDeviceTypes forPosition:(A
    if (@available(iOS 10.0, *)) {
        AVCaptureDeviceDiscoverySession *discoverySession = [AVCaptureDeviceDiscoverySession discoverySessionWithDeviceType
                                                                mediaType
                                                                position

        NSArray<AVCaptureDevice *> *sortedDiscoveredDevices = [discoverySession.devices sortedArrayUsingComparator:^NSCompa
        NSUInteger index1 = [sortedDeviceTypes indexOfObject:device1.deviceType];
        NSUInteger index2 = [sortedDeviceTypes indexOfObject:device2.deviceType];
        if (index1 < index2)
            return NSOrderedAscending;
        if (index1 > index2)
            return NSOrderedDescending;
        return NSOrderedSame;
    }];
    return sortedDiscoveredDevices.firstObject;
}
// under iOS 10.0
else {
    NSArray *devices = [AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo];
    for (AVCaptureDevice *device in devices)
        if (device.position == position)
            return device;

    return nil;
}
}

// sorted by priority (1st is the most desired device type)
- (NSArray<AVCaptureDeviceType> *)sortedDeviceTypes API_AVAILABLE(ios(10.0));
{
    if (@available(iOS 13.0, *)) {
        return @[AVCaptureDeviceTypeBuiltInTripleCamera,
                AVCaptureDeviceTypeBuiltInDualWideCamera,
                AVCaptureDeviceTypeBuiltInDualCamera,
                AVCaptureDeviceTypeBuiltInWideAngleCamera,
                AVCaptureDeviceTypeBuiltInUltraWideCamera,
                AVCaptureDeviceTypeBuiltInTelephotoCamera];
    }
    if (@available(iOS 10.2, *)) {
        return @[AVCaptureDeviceTypeBuiltInDualCamera,
                AVCaptureDeviceTypeBuiltInWideAngleCamera,
                AVCaptureDeviceTypeBuiltInTelephotoCamera];
    }
    // iOS 10.0
    else {
        return @[AVCaptureDeviceTypeBuiltInWideAngleCamera,
                AVCaptureDeviceTypeBuiltInTelephotoCamera];
    }
}
}

```

- Using virtual camera like *builtInTripleCamera* means that the system will start using ultra wide camera and you might not want to initially show ultra wide preview for the user. To fix this you have to raise the zoom factor. A raised zoom factor will make the system change to the standard/default camera. You might ask how the system will change back to ultra wide camera if we bring the phone too close to the object (under the minimum focus distance value). The phone has a proximity sensor that will make the system change to ultra wide camera despite of having raised zoom factor. To apply appropriate zoom factor the following code is recommended:

```

- (float) zoomFactorScale {
    if (@available(iOS 13.0, *)) {
        if (self.device &&
            (self.device.deviceType == AVCaptureDeviceTypeBuiltInTripleCamera ||
             self.device.deviceType == AVCaptureDeviceTypeBuiltInDualWideCamera))
        {
            NSArray<NSNumber*> *zoomFactors = [self.device virtualDeviceSwitchOverVideoZoomFactors];
            if (zoomFactors.count > 0)
                return zoomFactors.firstObject.floatValue;
            else
                return 1;
        }
        else
            return 1;
    }
}
}

```



