Designing a QR Code Reader Application

Developing an application to be used with an iOS device to read incoming QR Codes.

Author: Trevor Emerick

Date: 3/30/2015

Objective

This application note will show how to design an iOS application to look both appealing and have the ability to scan different QR Codes.

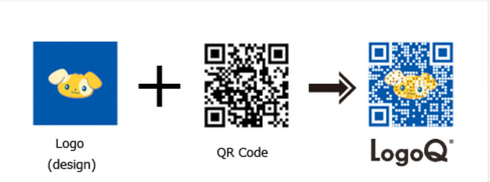
Introduction

This application note will walk you through the steps of creating a simple iOS application, which will then be taken a step further to successfully recognize a QR Code. Historically, these QR codes contain valuable information that can be obtained in seconds. Comparatively this attached information once obtained could be documented on paper. However, this becomes a problem in a fast paced environment when one does not have time to write down obtained information and thus has a higher chance of forgetting this information. QR codes can help rectify this problem. For this application, it will be assumed that you will have access to a Macintosh operating system, which is preloaded on Apple Computers, and Xcode, which is available through the Apple App Store.

Background

QR, also known as Quick Response, is a newer barcode design that has become more popular with the increase in mobile device sales. These codes can also be referred to as 2d barcodes or mobile codes. Many mobile device companies do not include a native QR reader app but they can be easily downloaded from their respective application stores. Though the use of QR is not as widely spread as UPC codes it is still gaining popularity in today’s society. Many of these QR codes can be seen as one goes throughout any given day. One can find them in various settings from the one referencing a local party happening this weekend, to renting a car in a big city like San Francisco, or on a resume to direct a recruiter to your LinkedIn profile. Unlike UPC codes, the QR code was designed for quick and efficient reading. These codes came to be in 1994 with the Denso Corporation. Many individuals within the company were very skeptical of the new code with the high usage of UPC codes in the product and grocery industry. Even with these doubts the QR codes were on the road to success.

The new QR codes look very different from the traditional UPC codes. Each QR code is based on a matrix barcode, which can be read by any mobile device with a camera. These codes are typically in a small white square box with black geometric shapes inside of it. This is by design and intential in that these codes, due to their unique design, are eye catching to those walking by. This has also aided in the increased use QR codes in ones day-to-day life. Unlike traditional UPC codes, QR codes have the ability to store a wide variety of information. This information usually would take longer to obtain without these quick response codes. The attributes that can be stored in these QR codes include phone numbers, websites, pictures, profiles, names, etc. If one was shown or asked what an UPC code can store, they would immediately think of price. The ability to have a variety of information comparatively available has increased the popularity of quick response codes.



When Denso first created the code it was considered to be one basic code, whose shapes could be manipulated slightly, thus making a new code that could then be produced. In reality, this design was then taken one step further to make multiple types. These types could be changed by their size, look, orientation or even background. For example, LogoQ is a type of QR code that uses the same fundamentals but has a background image that it is placed on top of. This process can be seen in the figure to the right. The iQR codes can either be square or rectangular. Once this was decided, the orientation of the code box can be changed to either horizontal or vertical. From a consumer standpoint, this was a very smart move. It allowed for the customization of QR codes based on the space available for placement or the look. This was just one of the innovative moves by the Denso Corporation in the early stages of QR Codes.

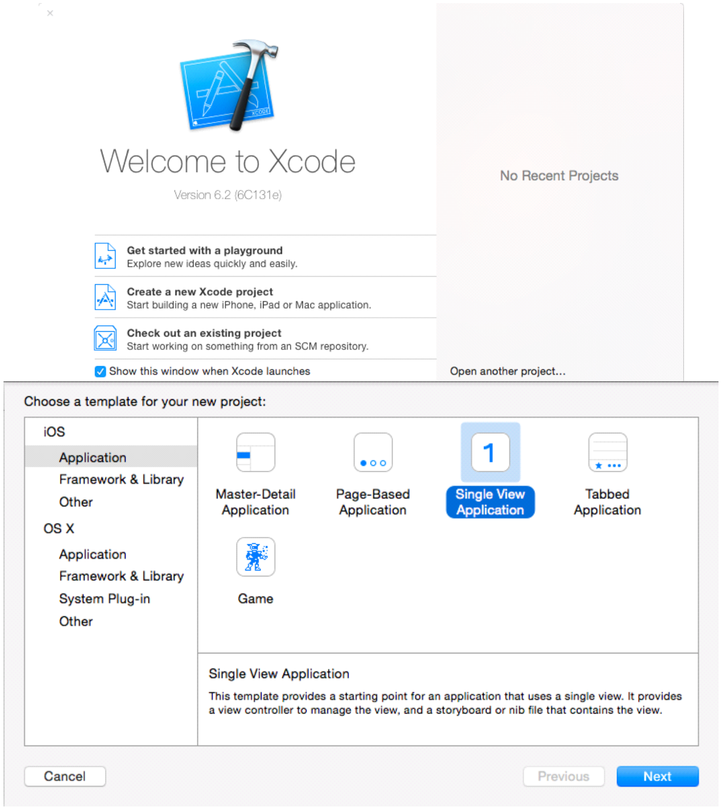
One industry that aided in the increased use of QR Codes was the automotive industry. The automotive inidustry mainly used these codes for the purpose of tracking processes in their electronic Kanban, which was a production management system. This included many tasks from producing all the way to shipping. While this may be very surprising to some, in actuality it is very counterintuitive. Before the use of QR codes these processes would be tracked via a paper trail. In the new digital age, one may realize that paper trails are very hard to follow and cumbersome. Using QR Codes in their electronic Kanban provided a more efficient workflow and easier to follow.

Quick Response codes have become very popular in other countries such as Japan however have been slow to come to popularity in the United States. This is mainly due to the ease of access to information through social media. Instead of looking for party information on a poster, many people look for an invite on Facebook. Although the increased use of social media in this country does not help with increasing the popularity of the QR codes here in America, there is, a potentially large use for it in industry. For companies that have a large number of visitors, contractors and distributors, the use of QR codes can be very beneficial in their day to day operations. Vast majorities of people have smartphones that get email. If the company can email their visitors, contractors and distributors the QR code prior to their arrival, it can then be scanned at the entrance, product pick up location, and on exit. This provides peace of mind for the business with who is on their premise as well as getting the correct shipment to the correct contractor or distributor. Thus it is a win win on both sides of the manufacturing production system. Efficiency for the contractors and distributors and accuracy on the side of the manufacturer. These codes will start appearing more and more in different industries at the distribution and manufacturing levels of the product channels.

Using Xcode to Develop a QR Code Reader for iOS

Building the Demo

For those that do not have any previous experience with writing iOS applications, you will need to obtain a copy of Xcode from the Macintosh App Store. Once the application has finished downloading and installing, navigate to the application folder in finder. Then click on the newly installed Xcode icon that will launch it. Once it has finished loading you will see the window displayed above. You will want to select the second option, **Create a new Xcode Project**. After selecting this, you will be directed to the next window that will ask you to choose a template for your new project.



Welcome to Xcode

Version 6.2 (6C131e)

Get started with a playground
Explore new ideas quickly and easily.

Create a new Xcode project
Start building a new iPhone, iPad or Mac application.

Check out an existing project
Start working on something from an SCM repository.

☑ Show this window when Xcode launches

No Recent Projects

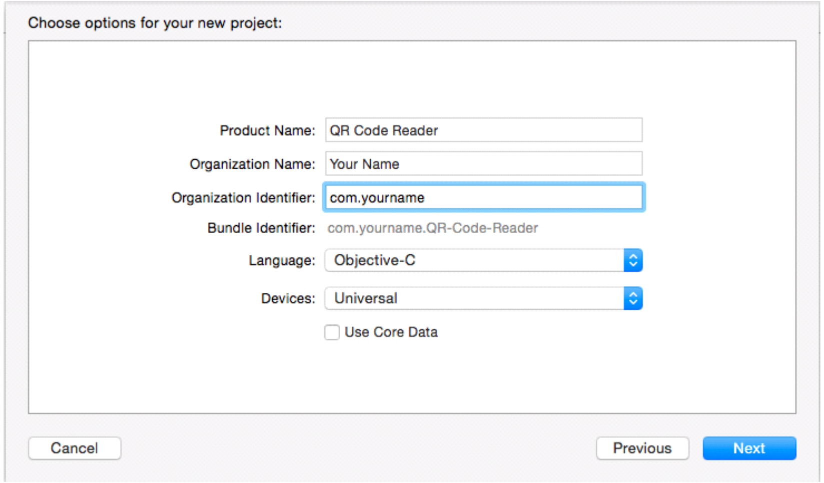
Open another project...

Choose a template for your new project:

Single View Application

This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a storyboard or nib file that contains the view.

When using a QR code reader, there is a loading screen and then a single view that you see when using the application. Therefore out of all of the options we want to choose the **Single View Application.** After being chosen, a **Product Name, Organization Name, and Organization Identifier** will need to be entered. These can be seen below as a guide as to what your screen should look like.

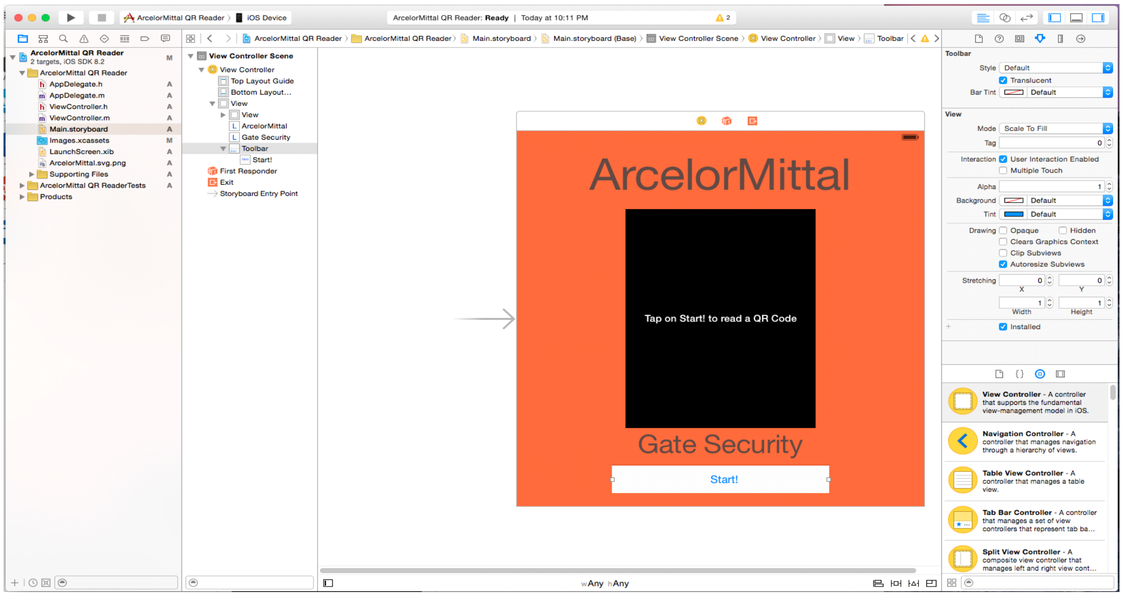


Choose options for your new project:

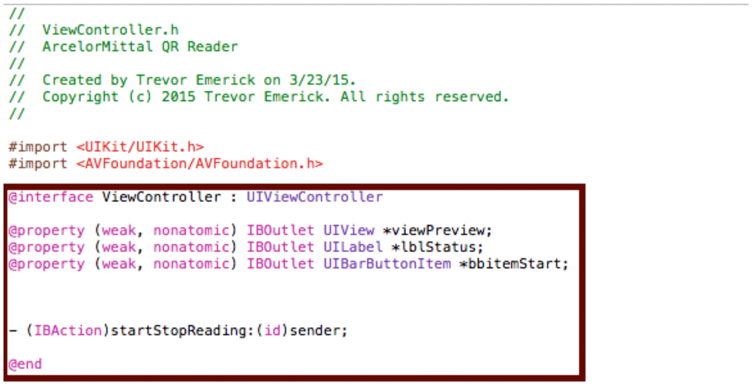
Product Name: QR Code Reader
Organization Name: Your Name
Organization Identifier: com.yourname
Bundle Identifier: com.yourname.QR-Code-Reader
Language: Objective-C
Devices: Universal
☐ Use Core Data

Setting Up the User Interface

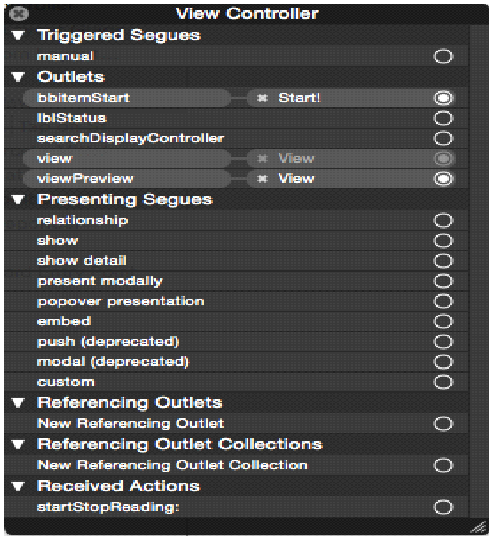
You will first want to navigate to the **Main.storyboard** file under the QR Code Reader folder. This will allow us to design what the main page of the app will look at. Under the third option, which can be seen in the photo to the left, we will want to add a **Start Button, Title, Second View Window, and a Tap to Start message.**
Once these are added you should have something that looks similar to the image below. The beauty of Xcode is that it has auto adjusting built in for use on different iOS device screens without having to develop multiple application versions for each.



Now that we have a layout that can be used by a user we will want to modify it, and provide it with functionality. Next, you want to navigate to the **ViewController.h**. We will need to add IBOutlet methods that can then be connected to the functions that were created on the **Main.storyboard**. You will want to add the following lines of code to the **ViewController.h** so that we have these functionalities. The last line before @end provides an action method that will allow for the start and stop of the video capturing.



```
//
//  ViewController.h
//  ArcelorMittal QR Reader
//
//  Created by Trevor Emerick on 3/23/15.
//  Copyright (c) 2015 Trevor Emerick. All rights reserved.
//

#import <UIKit/UIKit.h>
#import <AVFoundation/AVFoundation.h>

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UIView *viewPreview;
@property (weak, nonatomic) IBOutlet UILabel *lblStatus;
@property (weak, nonatomic) IBOutlet UIBarButtonItem *bbitemStart;


- (IBAction)startStopReading:(id)sender;

@end
```

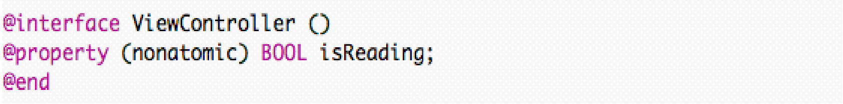


View Controller
- Triggered Segues
  - manual ○
- Outlets
  - bbitemStart — Start! ●
  - lblStatus ○
  - searchDisplayController ○
  - view — View ●
  - viewPreview — View ●
- Presenting Segues
  - relationship ○
  - show ○
  - show detail ○
  - present modally ○
  - popover presentation ○
  - embed ○
  - push (deprecated) ○
  - modal (deprecated) ○
  - custom ○
- Referencing Outlets
  - New Referencing Outlet ○
- Referencing Outlet Collections
  - New Referencing Outlet Collection ○
- Received Actions
  - startStopReading: ○

Next, go back to the **Main.storyboard** and right click on the view controller. You will see a pop up like the one below. Now connect the **bbitemStart** to the **Start** button that was created in the first steps. Now the view controller should look like the one on the right.

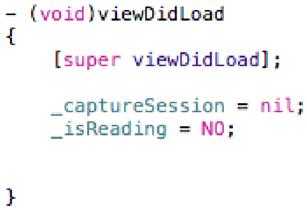
QR Code Reader

Since we now have the general design and interface. the next step is to implement the actual reading of the QR Codes. The most beneficial way to start this process is to use the IBAction that we wrote earlier which consisted of **StartStopReading**. This is already declared and connected to the start bar. In the **ViewController.m** file you will want to add the following flags in the private interface section.



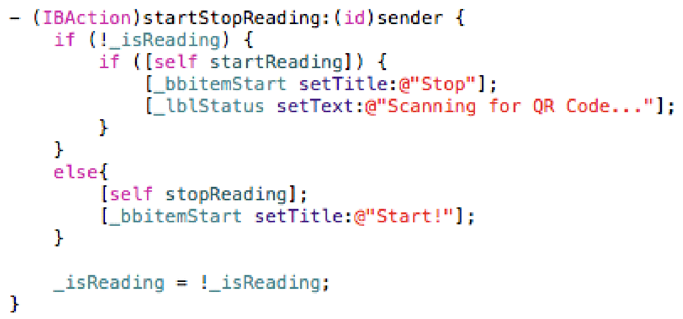
```
@interface ViewController ()
@property (nonatomic) BOOL isReading;
@end
```

The second line of code is important to note, and it may be obvious when looking at the statement **isReading**. If the value of the Boolean statement is **No** the application is not scanning for a code. The opposite of this is true when it becomes **yes**. In the next step, you will want to give the variables an initial value. In this case we want to make **isReading** equal to no so that it is not looking for a barcode everytime on launch. This initial declaration flag can be initiated inside the **viewDidLoad**, which can be seen below.



```
- (void)viewDidLoad
{
    [super viewDidLoad];

    _captureSession = nil;
    _isReading = NO;


}
```

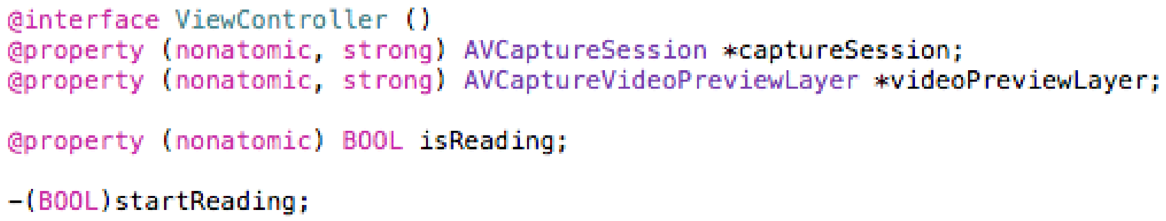
After this has been complete the next thing we want to do is implement the **startStopReading** using the IBAction method. In order to do this we will want to add the following lines of code.



```
- (IBAction)startStopReading:(id)sender {
    if (!_isReading) {
        if ([self startReading]) {
            [_bbitemStart setTitle:@"Stop"];
            [_lblStatus setText:@"Scanning for QR Code..."];
        }
    }
    else{
        [self stopReading];
        [_bbitemStart setTitle:@"Start!"];
    }

    _isReading = !_isReading;
}
```

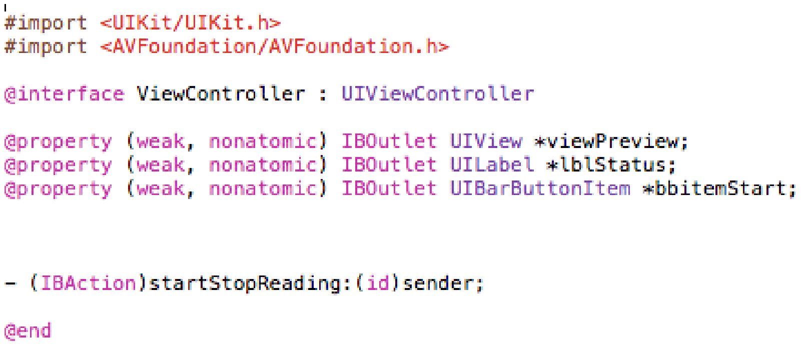
The implementation of this code is to first check if the app is already reading. If it is true that the application is currently not reading a QR code, we will want to start the **startReading** process. If this is the case we will switch the title of the current **Start** button to **Stop** as well as the status message. Another instance of this is if the application is already scanning a code, the program will jump to the else statement. This will then set the button to be start. It does not matter what value the **isReading** currently has, but at the end we will want to set its value to the exact opposite of what it is currently.

The next set of values we will want to add will be one of the most important things that can be added to the top private section of the **ViewController.m**. These lines of code are critical to the **startReading** portion of the application.



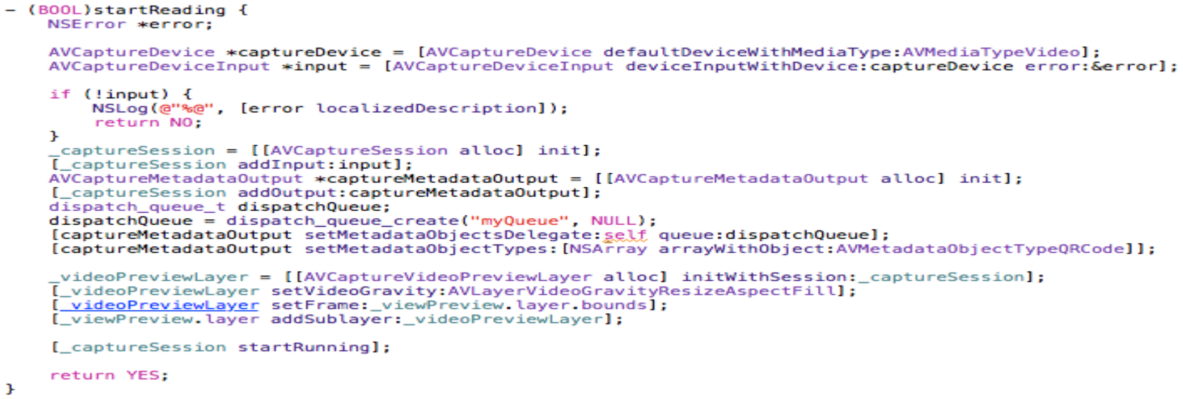
```
@interface ViewController ()
@property (nonatomic, strong) AVCaptureSession *captureSession;
@property (nonatomic, strong) AVCaptureVideoPreviewLayer *videoPreviewLayer;

@property (nonatomic) BOOL isReading;

-(BOOL)startReading;
```

After completing this step, Xcode may complain about the **AVCaptureSession** and the **AVCaptureVideoPreviewLayer** classes. The reason for this is that the AV Foundation framework has not yet been imported into the program. This is the reasoning that we have added **_captureSession = nil;** in the private section of the m-file earlier. Please check that you have added this. To correct this problem, you will want to navigate into the **ViewController.h** file and add the following line of code **#import <AVFoundation/AVFoundation.h>** under **#import <UIKit/UIKit.h>**. You should now not receive any errors about the compilier not being able to implement the **AVFoundationFramework**. While we are in the **ViewController.h** file we will want to add another line of could. You should now have a **ViewController.h** file that looks like the one below.



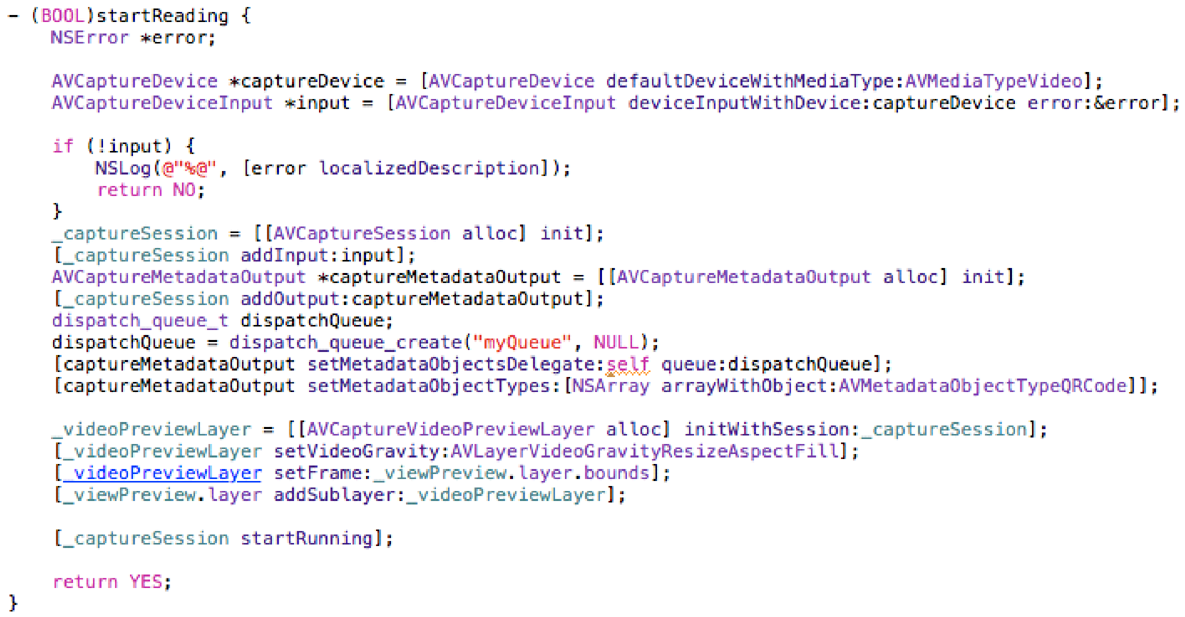
```
#import <UIKit/UIKit.h>
#import <AVFoundation/AVFoundation.h>

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UIView *viewPreview;
@property (weak, nonatomic) IBOutlet UILabel *lblStatus;
@property (weak, nonatomic) IBOutlet UIBarButtonItem *bbitemStart;



- (IBAction)startStopReading:(id)sender;

@end
```

Now that we have many of the application logistics figured out and imported we will now want to begin creating the **StartReading** and **StopReading** Booleans. This will be done back in the **ViewController.m** file. We will want to add the following lines of code to the m-file which will be used in the **StartReading** operation. We will first want to declare the **AVMediaTypeVideo** option for the media type with the **NSError** ability Built into the code. The next step will be to intiate the **CaptureSession**, by giving it the ability to works as an input and an output device. Adding the input and output attributes to the **captureSession** does this**.** The next lines of code, which uses the **AVCaptureMetadataOutput**, and the line after it, will take the QR code that has been read and convert it into something that a human using the application can interpret. This could be a url, phone number, or address. The **dispatch_queue_t** will take the scanned QR code and create "**myQueue**" in order for total use by the application tasks. Then the following metadata calls are used for finding the specific information that we are looking for from the QR code. Up until this point your m-file should look similar to that of the one below. If there are any errors you will want to fix them now.



```
- (BOOL)startReading {
    NSError *error;

    AVCaptureDevice *captureDevice = [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];
    AVCaptureDeviceInput *input = [AVCaptureDeviceInput deviceInputWithDevice:captureDevice error:&error];

    if (!input) {
        NSLog(@"%@", [error localizedDescription]);
        return NO;
    }
    _captureSession = [[AVCaptureSession alloc] init];
    [_captureSession addInput:input];
    AVCaptureMetadataOutput *captureMetadataOutput = [[AVCaptureMetadataOutput alloc] init];
    [_captureSession addOutput:captureMetadataOutput];
    dispatch_queue_t dispatchQueue;
    dispatchQueue = dispatch_queue_create("myQueue", NULL);
    [captureMetadataOutput setMetadataObjectsDelegate:self queue:dispatchQueue];
    [captureMetadataOutput setMetadataObjectTypes:[NSArray arrayWithObject:AVMetadataObjectTypeQRCode]];

    _videoPreviewLayer = [[AVCaptureVideoPreviewLayer alloc] initWithSession:_captureSession];
    [_videoPreviewLayer setVideoGravity:AVLayerVideoGravityResizeAspectFill];
    [_videoPreviewLayer setFrame:_viewPreview.layer.bounds];
    [_viewPreview.layer addSublayer:_videoPreviewLayer];

    [_captureSession startRunning];

    return YES;
}
```

Now that we have configured the output for the metadata, the next thing we want to do is provide a preview for the user of what they are aiming their camera at. Using the **videoPreviewLayer** from the **AVFoundations Framework** can do this. And then use the last line of code in order to start the session caputuring. Your **startReading** section of your m-file should now look like the one below.



```
- (BOOL)startReading {
    NSError *error;

    AVCaptureDevice *captureDevice = [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];
    AVCaptureDeviceInput *input = [AVCaptureDeviceInput deviceInputWithDevice:captureDevice error:&error];

    if (!input) {
        NSLog(@"%@", [error localizedDescription]);
        return NO;
    }
    _captureSession = [[AVCaptureSession alloc] init];
    [_captureSession addInput:input];
    AVCaptureMetadataOutput *captureMetadataOutput = [[AVCaptureMetadataOutput alloc] init];
    [_captureSession addOutput:captureMetadataOutput];
    dispatch_queue_t dispatchQueue;
    dispatchQueue = dispatch_queue_create("myQueue", NULL);
    [captureMetadataOutput setMetadataObjectsDelegate:self queue:dispatchQueue];
    [captureMetadataOutput setMetadataObjectTypes:[NSArray arrayWithObject:AVMetadataObjectTypeQRCode]];

    _videoPreviewLayer = [[AVCaptureVideoPreviewLayer alloc] initWithSession:_captureSession];
    [_videoPreviewLayer setVideoGravity:AVLayerVideoGravityResizeAspectFill];
    [_videoPreviewLayer setFrame:_viewPreview.layer.bounds];
    [_viewPreview.layer addSublayer:_videoPreviewLayer];

    [_captureSession startRunning];

    return YES;
}
```

If you were to now compile the application with Xcode and then load it onto your phone, you would see a preview when you hit the start button. Since we have not written the **stopReading** portion, do not expect to be able to read any QR codes or stop the preview without closing the application entirely. The next session will focus on using the metadata.

The first thing we will want to do when implementing the **NSArray** with the metadata is to make sure the array is not equal to nil. When reading a QR code, we are interested in the first part of the metadata. But once obtained we want to verify that this is the type of metadata that we are looking for. So when the application returns a true value, this is the data that we will want to read. Therefore it will cycle through what our **if** statement contains and process it. The following is what our **CaptureOutput** should look like for outputting what the QR code represents.

```
-(void)captureOutput:(AVCaptureOutput *)captureOutput didOutputMetadataObjects:(NSArray *)metadataObjects
fromConnection:(AVCaptureConnection *)connection{

   if (metadataObjects != nil && [metadataObjects count] > 0) {
      AVMetadataMachineReadableCodeObject *metadataObj = [metadataObjects objectAtIndex:0];
      if ([[metadataObj type] isEqualToString:AVMetadataObjectTypeQRCode]) {

         [_lblStatus performSelectorOnMainThread:@selector(setText:) withObject:[metadataObj stringValue] waitUntilDone:NO];

         [self performSelectorOnMainThread:@selector(stopReading) withObject:nil waitUntilDone:NO];
         [_bbitemStart performSelectorOnMainThread:@selector(setTitle:) withObject:@"Start!" waitUntilDone:NO];

         _isReading = NO;

         }
      }
   }

@end
```

With a working reader and preview that can process metadata, a **stopReading** can be used to bring the application to a halt. First we will want to declare it in the private section of our m-file.

After doing this we will implement what the **stopReading** function does. For the private section you will want to add the following of code, highlighted below.

```
@interface ViewController ()
...
...
-(void)stopReading;
@end
```

Below **startReading** outside of the private section we can now implement the **stopReading** functionality to the application. The main function of the tasks within **stopReading** is to end the processes that are being used by **startReading**. The capture session will go from currently reading back to nil. We will then want to remove the preview sessions and end the **captureSession**. These commands can be seen below and should be added into your code.

```
-(void)stopReading{
   [_captureSession stopRunning];
   _captureSession = nil;

   [_videoPreviewLayer removeFromSuperlayer];
}
```

After this you are now ready to compile the finished code, upload it to your iOS device. Once these steps you are now ready to start scanning!