Foxit

Developer Guide

Foxit® MobilePDF SDK

iOS

Microsoft® Partner
Gold Independent Software Vendor (ISV)

TABLE OF CONTENTS

1 Introduction to Foxit MobilePDF SDK

Have you ever worried about the complexity of the PDF specification? Or have you ever felt lost when asked to build a full-featured PDF app within a limited time-frame? If your answer is "Yes", then congratulations! You have just found the best solution in the industry for rapidly integrating PDF functionality into your apps.

1.1 Why Foxit MobilePDF SDK is your choice

Foxit is an Amazon-invested leading software provider of solutions for reading, editing, creating, organizing, and securing PDF documents. Foxit PDF SDK libraries have been used in many of today's leading apps, and they are proven, robust, and battle-tested to provide the quality, performance, and features that the industry's largest apps demand. Foxit MobilePDF SDK is a new SDK product which is developed for providing quick PDF viewing and manipulation support for mobile platforms. Customers choose it for the following reasons:

- **Easy to integrate**

Developers can seamlessly integrate Foxit MobilePDF SDK into their own apps with just a few lines of code.

- **Perfectly designed**

Foxit MobilePDF SDK is designed with a simple, clean, and friendly style, which provides the best user experience.

- **Flexible customization**

Foxit MobilePDF SDK provides the source code for the user interface which lets the developers have full control of the functionality and appearance of their apps.

- **Robust performance on mobile platforms**

Foxit MobilePDF SDK provides an OOM (out-of-memory) recovery mechanism to ensure the app has high robust performance when running the app on a mobile device which offers limited memory.

- **Powered by Foxit's high fidelity rendering PDF engine**

The core technology of Foxit MobilePDF SDK is based on Foxit's PDF engine, which is trusted by a large number of the world's largest and well-known companies. Foxit's powerful engine makes the app fast on parsing, rendering, and makes document viewing consistent on a variety of devices.

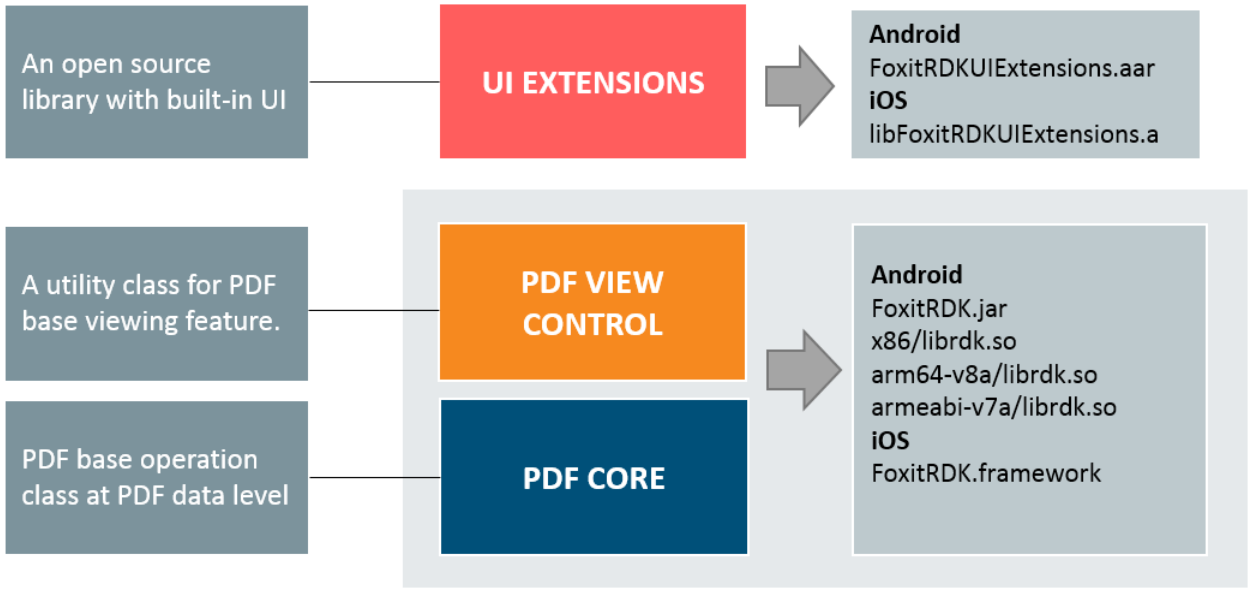
- **Premium World-side Support**

Foxit offers premium support for its developer products because when you are developing mission critical products you need the best support. Foxit has one of the PDF industry's largest team of support engineers. Updates are released on a regular basis to improve user experience by adding new features and enhancements.

1.2 Foxit MobilePDF SDK

Foxit MobilePDF SDK is a Rapid Development Kit for mobile platforms which focuses on helping developers easily integrate powerful Foxit PDF technology into their own apps. With Foxit MobilePDF SDK, even developers with a limited knowledge of PDF can quickly build a professional PDF viewer with just a few lines of code on iOS and Android platforms.

Foxit MobilePDF SDK consists of three elements as shown in the following picture.



The three elements for Foxit MobilePDF SDK

- **PDF Core API**

The PDF Core API is the heart of this SDK and is built on Foxit's powerful underlying technology. It provides the functionality for basic PDF operation features, and is utilized by the PDF View Control and UI Extensions Component, which ensures the apps can achieve high performance and efficiency. The Core API can be used independently for document rendering, analysis, text extraction, text search, form filling, digital signatures, Pressure Sensitive Ink, certificate and password security, annotation creation and manipulation and much more.

- **PDF View Control**

The PDF View Control is a utility class that provides the functionality for developers to interact with rendering PDF documents per their requirements. With Foxit's renowned and widely used PDF rendering technology at its core, the View Control provides fast and high quality rendering, zooming, scrolling and page navigation features. The View Control derives from platform related viewer classes (e.g. UIView on iOS and Android.View.ViewGroup on Android) and allows for extension to accommodate specific user needs.

- **UI Extensions Component**

The UI Extensions **Component** is an open source library that provides a customizable user interface with built-in support for text selection, markup annotation, outline navigation, reading bookmarks, full-text searching, form filling, text reflow, attachment, digital/handwritten signature, reflow, document editing and password encryption. These features in the UI Extensions Component are implemented using the PDF Core API and PDF View Control. Developers can utilize these ready-to-use UI implementations to build a PDF viewer quickly with the added benefit of complete flexibility and control to customize the UI design as desired.

1.3 Key features

Foxit MobilePDF SDK has several main features which help app developers quickly implement the functions that they really need and reduce the development cost.

Features	
PDF Document	Open and close files, set and get metadata.
PDF Page	Parse, render, read, and edit PDF pages.
Render	Graphics engine created on a bitmap for platform graphics device.
Reflow	Rearrange page content.
Text Select	Select text in a PDF document.
Text Search	Search text in a PDF document.
Outline	Directly locate and link to point of interest within a document.
Reading Bookmark	Mark progress and interesting passages as users read.
Annotation	Create, edit and remove annotations.
Form	Fill form with JavaScript support, export and import form data by XFDF/FDF/XML file.

Signature	Sign a PDF document, verify a signature, add or delete a signature field.
Security	Protect PDFs with password or certificate.
Out of Memory	Recover from an OOM condition

Note *Outline is the technical term used in the PDF specification for what is commonly known as bookmarks in traditional desktop PDF viewers. Reading bookmarks are commonly used on mobile and tablet PDF viewers to mark progress and interesting passages as users read but are not technically outline and are stored at app level rather than within the PDF itself.*

1.4 Evaluation

Foxit MobilePDF SDK allows users to download trial version to evaluate SDK. The trial version has no difference from the standard licensed version except for the free 21-day trial limitation and the trial watermarks in the generated pages. After the evaluation period expires, customers should contact the Foxit sales team and purchase licenses to continue using Foxit MobilePDF SDK.

1.5 License

Developers should purchase licenses to use Foxit MobilePDF SDK in their solutions. Licenses grant developers permission to release their apps which utilize Foxit MobilePDF SDK. However, users are prohibited to distribute any documents, sample code, or source code in the released package of Foxit MobilePDF SDK to any third party without written permission from Foxit Software Incorporated.

1.6 About this Guide

Foxit MobilePDF SDK is currently available on iOS and Android platforms. This guide is intended for the developers who need to integrate Foxit MobilePDF SDK for iOS into their own apps. It aims at introducing the following sections:

- Section 1: gives an introduction of Foxit MobilePDF SDK.
- Section 2: illustrates the package structure, running demo, and adding PDF SDK into app.
- Section 3: introduces how to customize the UI implementation.
- Section 4: shows how to create a custom tool.
- Section 5: lists some frequently asked questions.
- Section 6: provides support information.

2 Getting Started

It is very easy to setup Foxit MobilePDF SDK and see it in action! It takes just a few minutes and we will show you how to use it on the iOS platform. The following sections introduce the structure of the installation package, how to run a demo, and how to create your own project in Xcode.

2.1 Requirements

- iOS 9.0 or higher
- Xcode 7.0 or newer for Objective-C; Xcode 8.0 or newer for Swift

2.2 What is in the Package

Download the "foxit_mobile_pdf_sdk_ios_en.zip" package, and extract it to a new directory like "foxit_mobile_pdf_sdk_ios_en" as shown in Figure 2-1. The package contains:

docs:	A folder containing API references, developer guide.
libs:	A folder containing license files, SDK framework, UI Extensions Component and source code.
samples:	A folder containing iOS sample projects.
getting_started_ios.pdf:	A quick guide for Foxit MobilePDF SDK for iOS.
legal.txt:	Legal and copyright information.
release_notes.txt:	Release information.

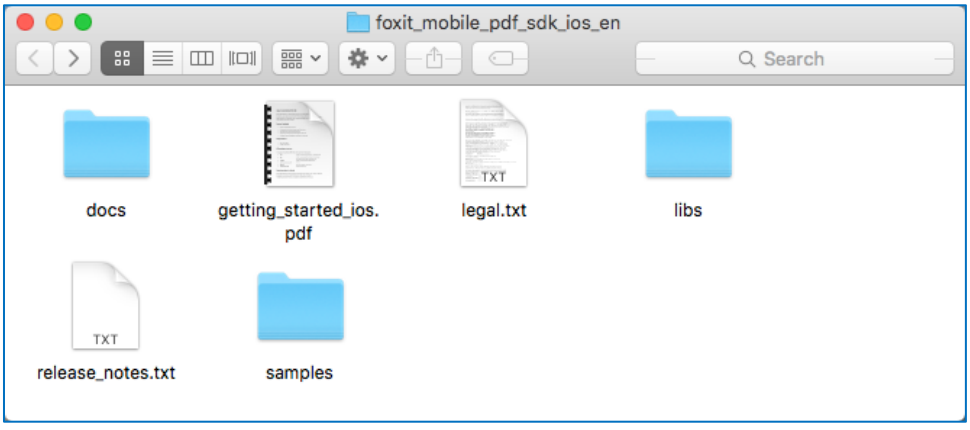


Figure 2-1

In the "libs" folder as shown in Figure 2-2, there are items that make up the core components of Foxit MobilePDF SDK for iOS.

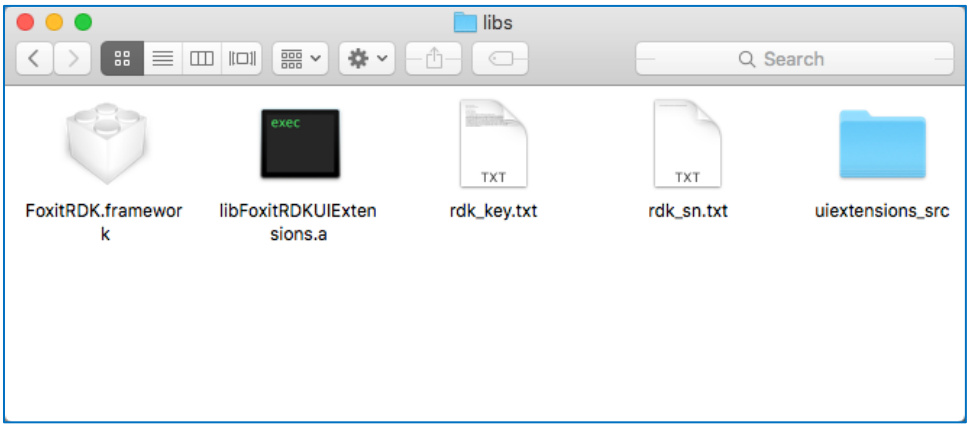


Figure 2-2

- ***FoxitRDK.framework*** – The framework that includes the Foxit MobilePDF SDK dynamic library and associated header files.
- ***libFoxitRDKUIExtensions.a*** – It's a universal static library (for both simulator and iOS device) generated by the "**uiextensions**" project found in the "libs/uiextensions_src" folder.
- ***uiextensions*** *project*- found in the "libs/uiextensions_src" folder. It is an open source library that contains some ready-to-use UI module implementations, which can help developers rapidly embed a fully functional PDF reader into their iOS app. Of course, developers are not forced to use the default UI, they can freely customize and design the UI for their specific apps through the "uiextensions" project.

2.3 How to run a demo

Download and install Xcode IDE (https://developer.apple.com/download/).

Note*: In this guide, we do not cover the installation of Xcode. You can refer to Apple's developer site if you haven't installed it already.*

Foxit MobilePDF SDK for iOS provides three useful demos in both Objective-C and Swift programming languages for developers to learn how to call the SDK. The Swift demos are located in the "swift" folder. (See Figure 2-3)

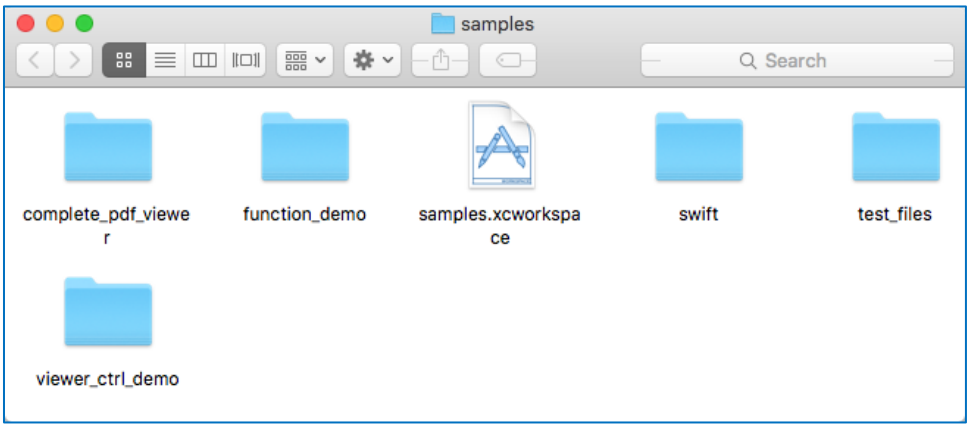


Figure 2-3

2.3.1 Function demo

The function demo is provided with Objective-C and Swift programming languages, which is used to show how to use Foxit MobilePDF SDK to realize some specific features related with PDF. This demo includes the following features:

- **pdf2txt**: extract all the text from a PDF document, and then save them to a text file.
- **outline**: edit outline (aka bookmark) appearances and titles.
- **annotation**: add some annotations to one page of a PDF.
- **docinfo**: export the basic information of a PDF to a text file.
- **render**: render a specific page of a PDF to a bitmap, and save the bitmap.
- **signature**: add a signature to PDF, sign PDF and verify the signature.

To run it in Xcode, follow the steps below:

a) Double-click **function_demo.xcodeproj** found in the "samples/function_demo" folder to open the demo in Xcode. (For Swift, double-click **function_demo_swift.xcodeproj** found in the "samples/swift/function_demo_swift" folder)

***Note**: There is another way to open the demo in Xocde: double-click **samples_xcworkspace** found in the "samples" folder. It is a workspace including the three demos.*

b) Click on "Product -> Run" to run the demo on an iOS device or simulator. In this guide, an iPhone 7 Simulator will be used as an example. After building the demo successfully, the features are listed like the Figure 2-4.

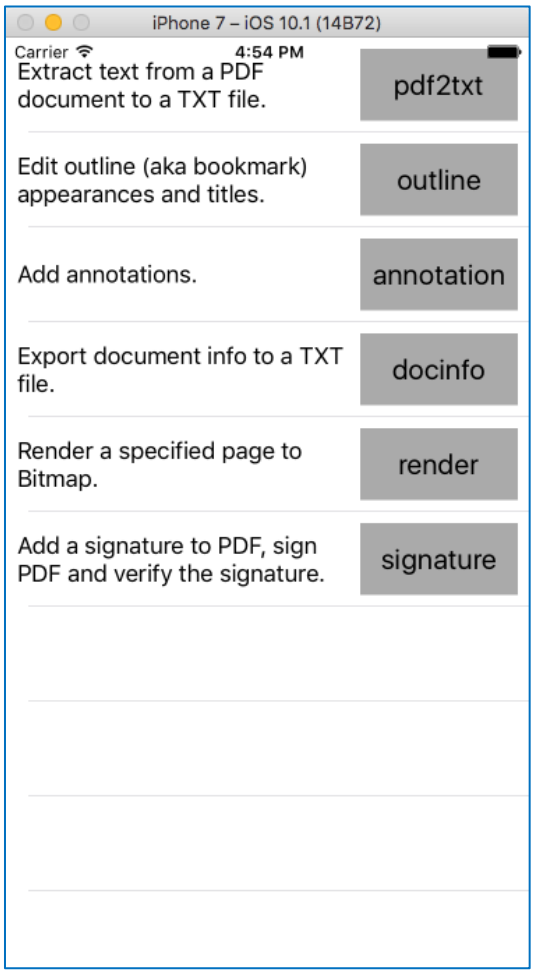


Figure 2-4

c) Click the feature buttons in the above picture to perform the corresponding actions. For example, click "pdf2txt", and then a message box will be popped up as shown in Figure 2-5. It shows where the text file was saved to. Just run the demo and try the features.

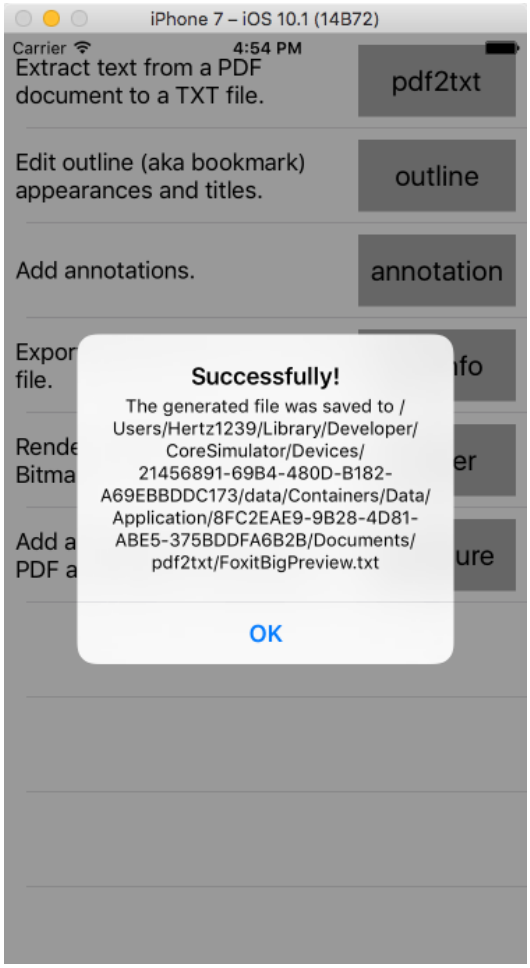


Figure 2-5

2.3.2 Viewer control demo

The viewer control demo is provided with Objective-C and Swift programming languages, which is used to demonstrate how to implement the features related to the View Control feature level, such as performing annotations (note, typewriter, highlight, underline, strikeout, squiggly, etc.), outline, reading bookmarks and text search. The logical structure of the code is quite clear and simple so that developers can quickly find the detailed implementation of features which are used widely in PDF apps, such as a PDF viewer. With this demo, developers can take a closer look at the APIs provided in Foxit MobilePDF SDK.

To run the demo in Xcode, please refer to the setup steps outlined in the Function demo.

Figure 2-6 shows what the demo looks like after it was built successfully. Here, an iPhone 7 Simulator will be used as an example to run the demo.

Figure 2-6

This demo provides the features like text search and listing reading bookmarks, outline and annotations. For example, click , select the second tab (outline), then the outline of this document will be displayed as shown in Figure 2-7.

Note *Outline is the technical term used in the PDF specification for what is commonly known as bookmarks in traditional desktop PDF viewers. Reading bookmarks are commonly used on mobile and tablet PDF viewers to mark progress and interesting passages as users read but are not technically outlines and are stored at app level rather than within the PDF itself.*

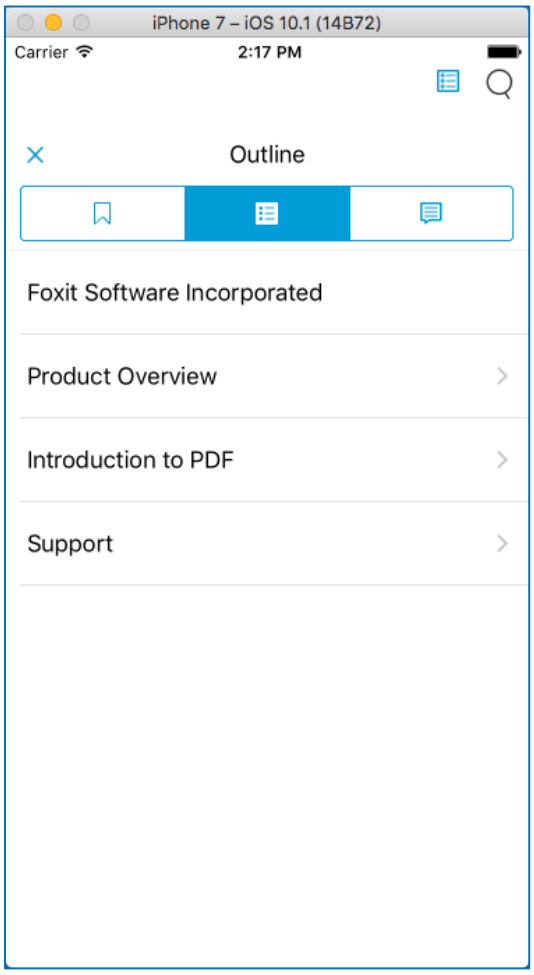


Figure 2-7

2.3.3 Complete PDF viewer demo

The complete PDF viewer demo demonstrates how to use Foxit MobilePDF SDK to realize a completely full-featured PDF viewer which is almost ready-to-use as a real world mobile PDF reader. This demo utilizes all of the features and built-in UI implementations which are provided in Foxit MobilePDF SDK.

To run the demo in Xcode, please refer to the setup steps outlined in the Function demo.

Here, an iPhone 7 Simulator will also be used as an example to run the demo. After building the demo successfully, on the start screen, choose a PDF file (e.g. "complete_pdf_viewer_guide_ios.pdf"), click it and then it will be opened and displayed as shown in Figure 2-8.

Note *If you want to use some other PDF files to test this demo, you need to put them onto the "Document" folder of the device.*

Figure 2-8

This demo realizes a completely full-featured PDF viewer, please feel free to run it and try it.

For example, it provides the page thumbnail feature. You can click the **View** menu, choose the **Thumbnail** as shown in Figure 2-9, and then the thumbnail of the document will be displayed as shown in Figure 2-10.


iPhone 7 – iOS 10.1 (14B72)
Carrier
6:48 PM
Foxit
PDF
Demo Guide
Foxit® Complete PDF Viewer
Single
Continuous
Thumbnail
Auto-Brightness
Reflow
Screen Lock

Figure 2-9

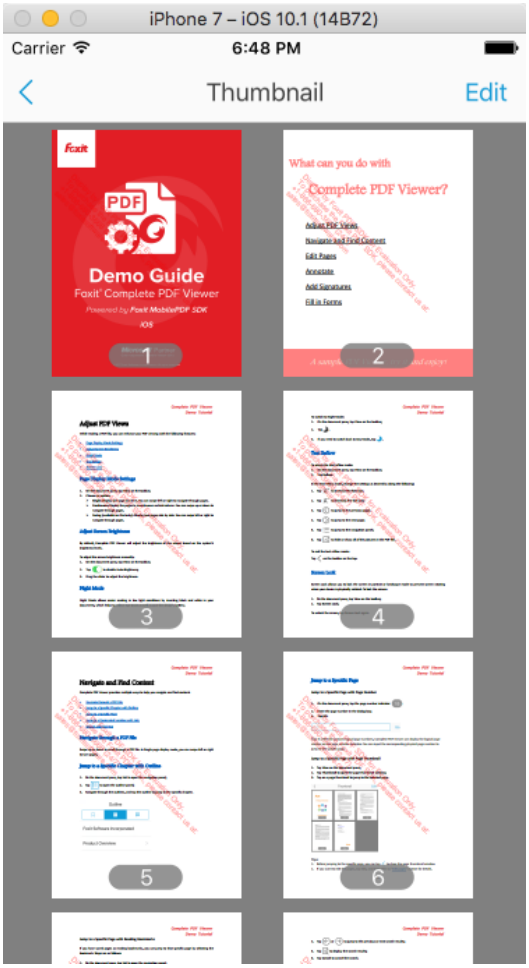


Figure 2-10

2.4 How to make an iOS app in Objective-C with Foxit MobilePDF SDK

This section will help you to quickly get started with using Foxit MobilePDF SDK to make an iOS app in Objective-C with step-by-step instructions provided. From now, you can get familiar with Foxit MobilePDF SDK and use Objective-C to create your first PDF iOS app in Xcode. This section includes the following steps:

1) Create a new iOS project in Objective-C
2) Integrate Foxit MobilePDF SDK into your apps
3) Apply the license key
4) Display a PDF document
5) Add support for Form Filling

6) Add support for Text Search

7) Add support for Annotations

2.4.1 Create a new iOS project in Objective-C

In this guide, we use Xcode 8.1 to create a new iOS project.

Fire up Xcode, choose **File** -> **New** -> **Project**..., and then select **iOS** -> **Single View Application** as shown in Figure 2-11. Click **Next**.

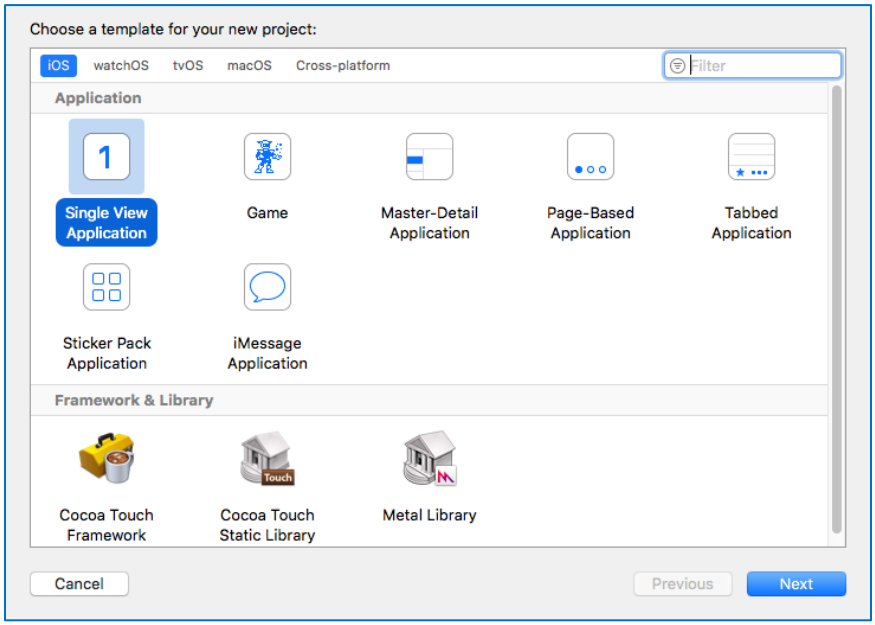


Figure 2-11

Choose the options for your new project as shown in Figure 2-12. Please make sure to choose Objective-C as the programming language. For simplicity, we don't check the Unit Tests and UI Tests which are used for automated testing. Then, Click **Next**.

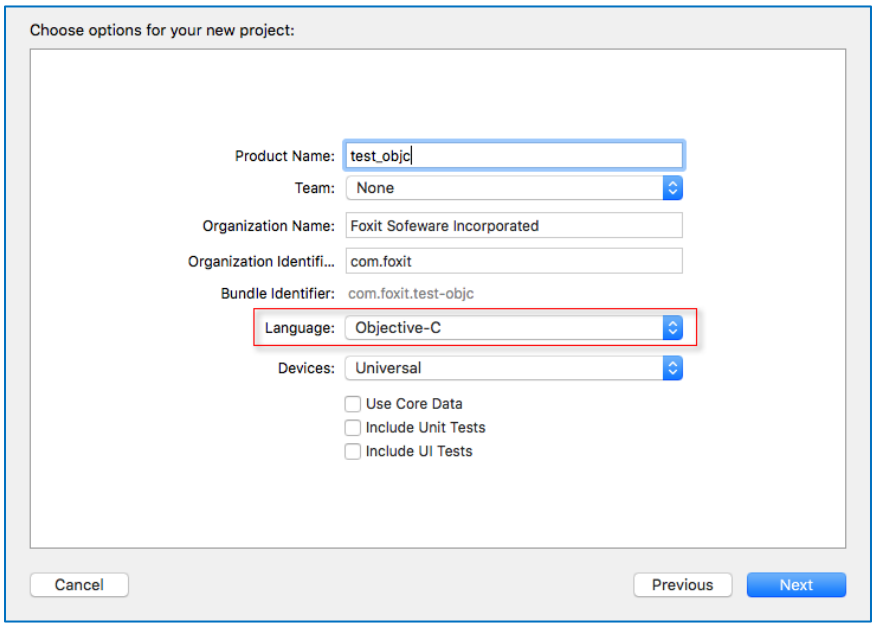


Figure 2-12

Place the project to the location as desired. The option "version control" is not actually important for building your first PDF app, so let's use the default setting. Here, we place the project to the desktop as shown in Figure 2-13. Then, click **Create**.

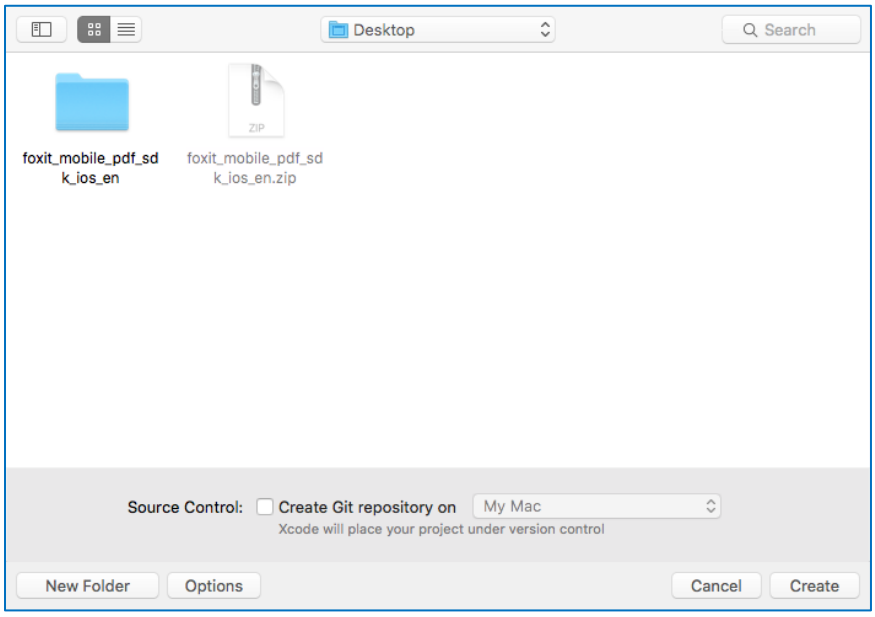


Figure 2-13

2.4.2 Integrate Foxit MobilePDF SDK into your apps

Note: *In this section, we will use the default built-in UI implementations to develop the app, for simplicity and convenience (use the UI Extensions Component directly, and don't need to build the source code project), we need to add the following files to the test_objc project.*

- ***FoxitRDK.framework*** - The framework that includes the Foxit MobilePDF SDK dynamic library and associated header files.
- ***libFoxitRDKUIExtensions.a*** – It's a universal static library (for both simulator and iOS device) generated by the "**uiextensions**" project found in the "libs/uiextensions_src" folder.

  Note *please keep in mind that you should include the corresponding header files for the classes you need to use in libFoxitRDKUIExtensions.a. Just find them in the "libs/uiextensions_src/uiextensions" folder.*
- ***Resource files*** – found in the "libs/uiextensions_src/uiextensions/resource" folder. They are needed for the default built-in UI implementations, such as images, strings and other resources.

Note: *The UI Extensions Component (**libFoxitRDKUIExtensions.a**) and resource files are not required for the section "Display a PDF document", so we just add "**FoxitRDK.framework**" to the test_objc project first. In the section "Add support for Form Filling", we will introduce how to add the UI Extensions Component and resource files.*

To add the dynamic framework "***FoxitRDK.framework***" into the *test_objc* project, please follows the steps below:

a) Right-click the "*test_objc*" project, select **Add Files to "test_objc"**… as shown in Figure 2-14.

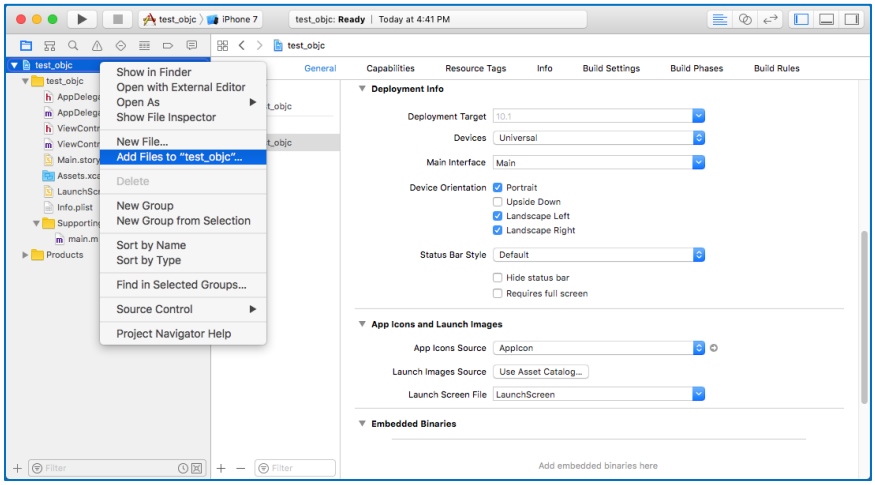


Figure 2-14

b) Find and choose "***FoxitRDK.framework***" in the "libs" folder of the download package, and then click **Add** as shown in Figure 2-15.

Note *Make sure to check the "**Copy items if needed**" option.*

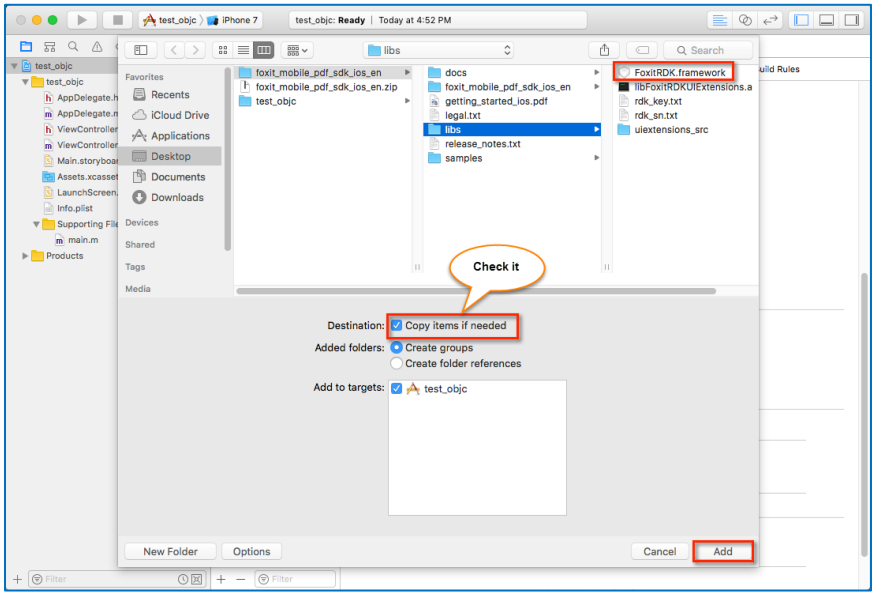


Figure 2-15

Then, the *test_objc* project will look like the Figure 2-16.

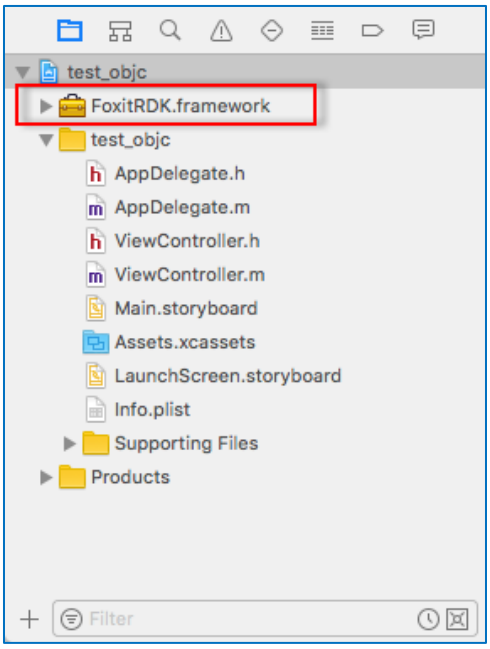


Figure 2-16

c) Add the dynamic framework "***FoxitRDK.framework***" to the Xcode's **Embedded Binaries**. Left-click the project, find **Embedded Binaries** in the **General** tab, and press on the **+** button as shown in Figure 2-17.

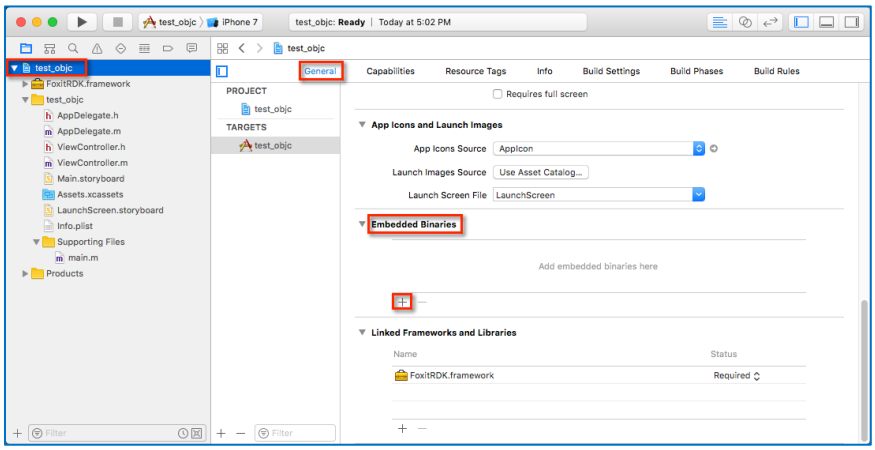


Figure 2-17

Then, choose the "***FoxitRDK.framework***" to add, and the **Embedded Binaries** will be like the Figure 2-18**.**

Figure 2-18

Now, we have added "***FoxitRDK.framework***" to the *test_objc* project successfully.

2.4.3 Apply the license key

It is necessary for apps to initialize and unlock Foxit MobilePDF SDK using a license before calling any APIs. The function *[FSLibrary init:sn key:key]* is provided to initialize Foxit MobilePDF SDK. The trial license files can be found in the "libs" folder of the download package. After the evaluation period expires, you should purchase an official license to continue using it. Finish the initialization in the didFinishLaunchingWithOptions method within the AppDelegate.m file.

```
#import "FoxitRDK/FSPDFObjC.h"

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {

    NSString* sn = @"";
```

```
    NSString* key =@"";
    enum FS_ERRORCODE eRet = [FSLibrary init:sn key:key];
    if (e_errSuccess != eRet) {
        return NO;
    }
    return YES;
}
```

Note *The parameter "sn" can be found in the "**rdk_sn.txt**" (the string after "SN=") and the "key" can be found in the "**rdk_key.txt**" (the string after "Sign=").*

Then we just have to call the *[FSLibrary release]* function to release the library in the applicationWillTerminate method.

```
- (void)applicationWillTerminate:(UIApplication *)application {
    [FSLibrary release];
}
```

In short, make sure that the AppDelegate.m file includes the following code:

```
#import "AppDelegate.h"
#import "FoxitRDK/FSPDFObjC.h"

@interface AppDelegate ()

@end

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {

    // The value of "sn" can be found in the "rdk_sn.txt".
    // The value of "key" can be found in the "rdk_key.txt".
    NSString* sn = @" ";
    NSString* key = @" ";

    enum FS_ERRORCODE eRet = [FSLibrary init:sn key:key];
    if (e_errSuccess != eRet) {
        return NO;
    }
    return YES;
}

- (void)applicationWillTerminate:(UIApplication *)application {
    [FSLibrary release];
}

@end
```

2.4.4 Display a PDF document

So far, we have added "***FoxitRDK.framework***" to the *test_objc* project, and finished the initialization of the Foxit MobilePDF SDK. Now, let's start building a simple PDF viewer with just a few lines of code.

Note: *The UI Extensions Component is not required if you only need to display a PDF document.*

First of all, add a PDF file to the project which will be used as the test file. For example, we use "Sample.pdf" found in the "samples\test_files" folder of the download package. Right-click the *test_objc* project, and select **Add Files to "test_objc"**... to add this file. After adding, you can see the PDF in the Xcode's **Copy Bundle Resources** as shown in Figure 2-19.

Note *You can add the PDF to* ***Copy Bundle Resources*** *directly. Just left-click the test_objc project, find* ***Copy Bundle Resources*** *in the* ***Build Phases*** *tab, press on the* ***+*** *button, and choose the file to add. You can refer to any PDF file, just add it to the Xcode's* ***Copy Bundle Resources****.*

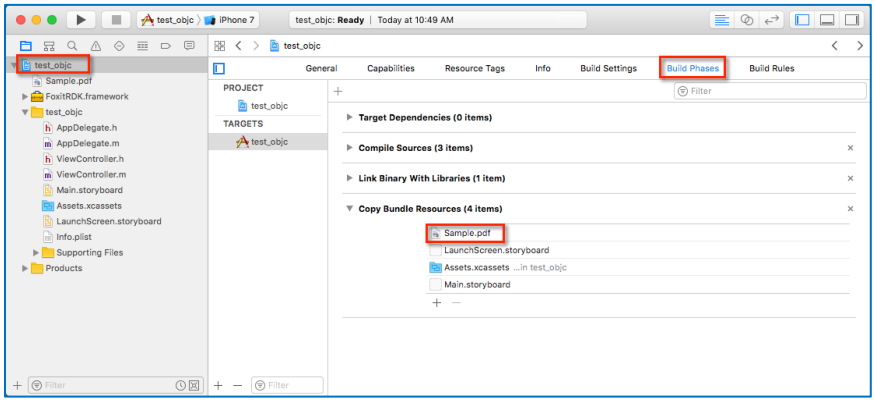


Figure 2-19

Then, add the following code to ViewController.m to display a PDF document. It's really easy to present a PDF on screen. All you need is to create a **FSPDFDoc** object and then show it with a **FSPDFViewCtrl** object.

Update ViewController.m as follows:

```
#import "ViewController.h"
#import "FoxitRDK/FSPDFViewControl.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    // Get the path of a PDF
    NSString* pdfPath = [[NSBundle mainBundle] pathForResource:@"Sample" ofType:@"pdf"];

    // Initialize a PDFDoc object with the path to the PDF file
    FSPDFDoc* pdfdoc = [FSPDFDoc createFromFilePath:pdfPath];
    if(e_errSuccess != [pdfdoc load:nil]) {
```

```
        return;
    }

    // Initialize a FSPDFViewCtrl object with the size of the entire screen
    FSPDFViewCtrl* pdfViewCtrl;
    pdfViewCtrl = [[FSPDFViewCtrl alloc] initWithFrame: [self.view bounds]];

    // Set the document to display
    [pdfViewCtrl setDoc:pdfdoc];

    // Add the pdfViewCtrl to the root view
    [self.view addSubview:pdfViewCtrl];

}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

Fantastic! We have now finished building a simple iOS app which uses Foxit MobilePDF SDK to display a PDF document with just a few lines of code. The next step is to run the project on a device or simulator.

In this guide, we build and run the project on an iPhone 7 Simulator, and you will see that the "Sample.pdf" document is displayed as shown in Figure 2-20. Now, this sample app has some basic PDF features, such as zooming in/out and page turning. Just have a try!

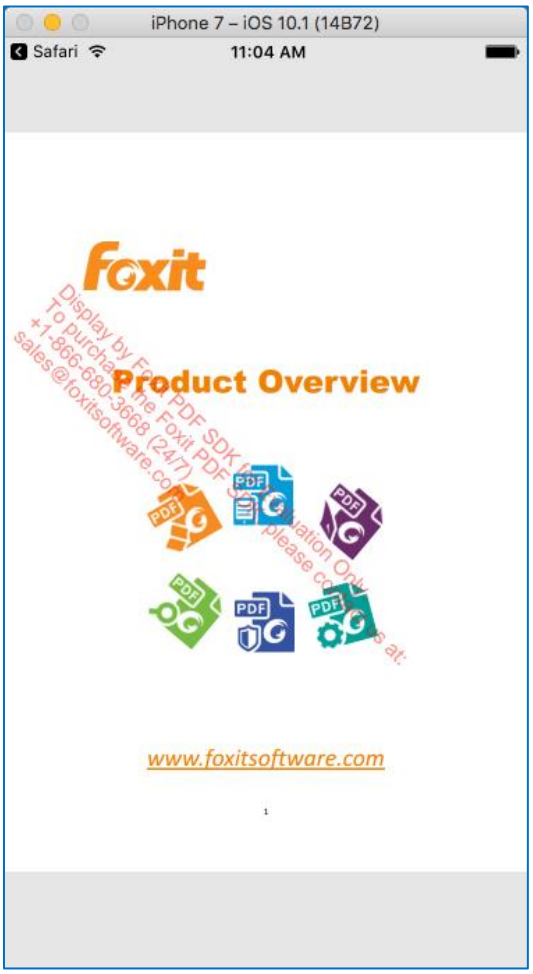


Figure 2-20

2.4.5 Add support for Form Filling

Foxit MobilePDF SDK comes with built-in support for features such as annotations, text search, outline and form filling. These visual features are implemented using Foxit MobilePDF SDK API and are shipped in the UI Extensions Component.

The form filling feature is already provided in the UI Extensions Component. It's simple and easy to integrate it into your app. For annotations and text search support, you can refer to the following sections.

In the previous sections, we have introduced how to add Foxit MobilePDF SDK to a project and how to build a simple iOS app for displaying a PDF document in Objective-C. Now, let’s extend the simple app (*test_objc*) further to learn how to add support for form filling.

First, let's do a test. Prepare a PDF form file, and add it to the *test_objc* project. For example, we use a PDF form file called "FoxitForm.pdf" found in the "samples/test_files" folder of the download package.

Open "ViewController.m", only change the file name from "Sample.pdf" to "FoxitForm.pdf" as follows:

```
NSString* pdfPath = [[NSBundle mainBundle] pathForResource:@"FoxitForm" ofType:@"pdf"];
```

Then rebuild and run the project, and you will see that the "FoxitForm.pdf" is displayed as shown in Figure 2-21. Now, it is just like a normal PDF file, in which the form fields cannot be edited.

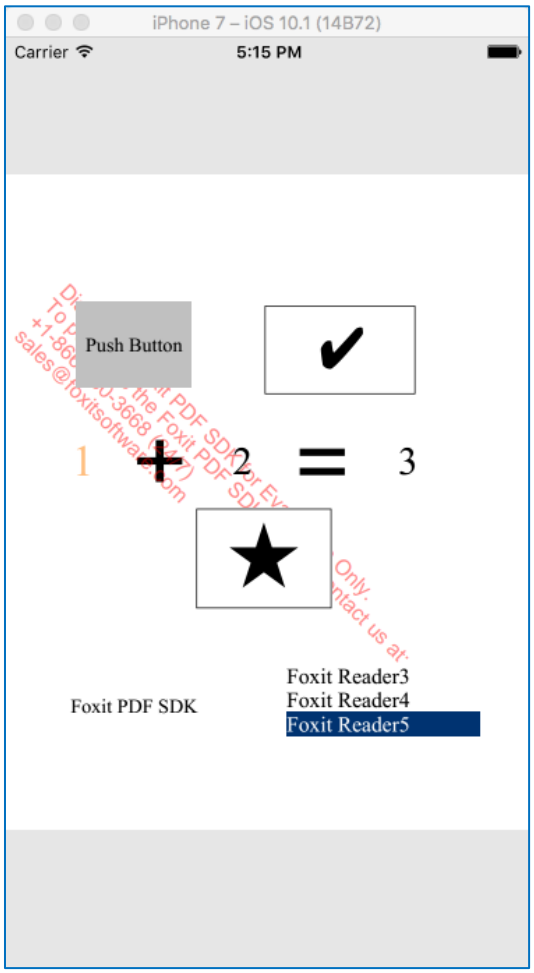


Figure 2-21

Next, let's add support for form filling. It's extremely easy! The key point is to instantiate a UIExtensionsManager object and set it to PDFViewCtrl. Please do the following preparatory work first, and then add code to finish the support for form filling.

Preparatory work

Step 1: Add UI Extensions Component (***libFoxitRDKUIExtensions.a***) to the project.

Note *In this app, we use the default built-in UI implementations to develop it, for simplicity and convenience, we will add* ***libFoxitRDKUIExtensions.a*** *to the test_objc project.*

Right-click the *test_objc* project, and select **Add Files to "test_objc"**… to add the extensions library. After adding, you can see the library in the Xcode's **Link Binary With Libraries** as shown in Figure 2-22.

Note *You can add the library to* ***Link Binary With Libraries*** *directly. Just left-click the test_objc project, find* ***Link Binary With Libraries*** *in the* ***Build Phases*** *tab, press on the* ***+*** *button, and choose the file to add. In either case, please check the "****Copy items if needed****" option when choosing the file.*

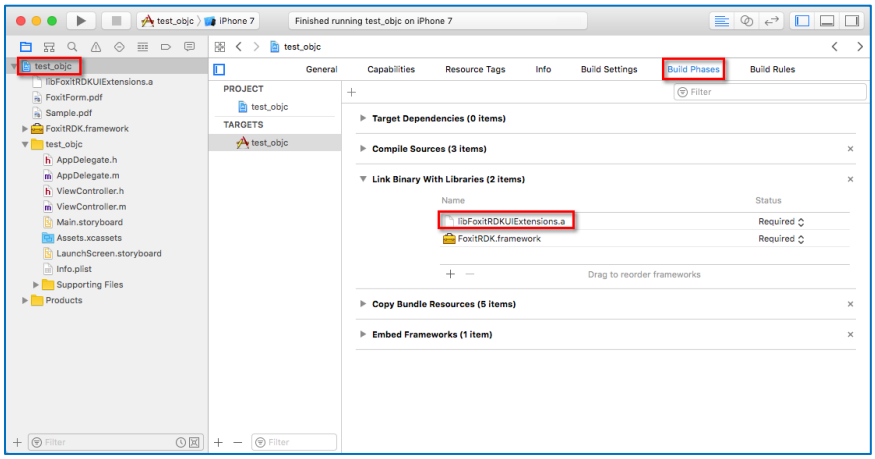


Figure 2-22

Step 2: Add "***-force_load libFoxitRDKUIExtensions.a -lstdc++***" to **Other Linker Flags** in the **Build Settings** tab as shown in Figure 2-23. "***-force_load libFoxitRDKUIExtensions.a"*** is used to load all of the members that implement any Objective-C class or category in the static library. "***-lstdc++***" ensures that the C++ standard library that is required by Foxit MobilePDF SDK will be included at link time.

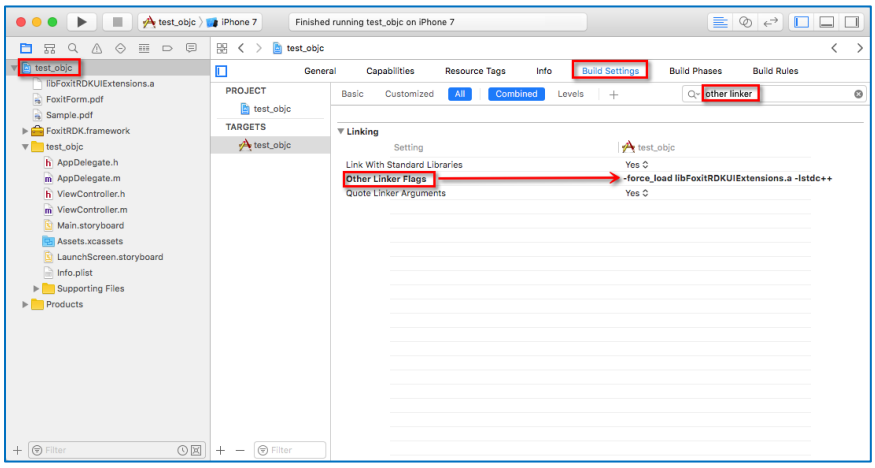


Figure 2-23

Step 3: Copy the **uiextensions** folder from the "libs/uiextensions_src" of the download package to "test_objc". This file contains the header files for ***libFoxitRDKUIExtensions.a***. Then, the *"test_objc"* folder will look like Figure 2-24.

Note *This project only needs the "UIExtensionsManager.h" file. So you can just add this header file found in the "libs/uiextensions_src/uiextensions" of the download package to the project.*

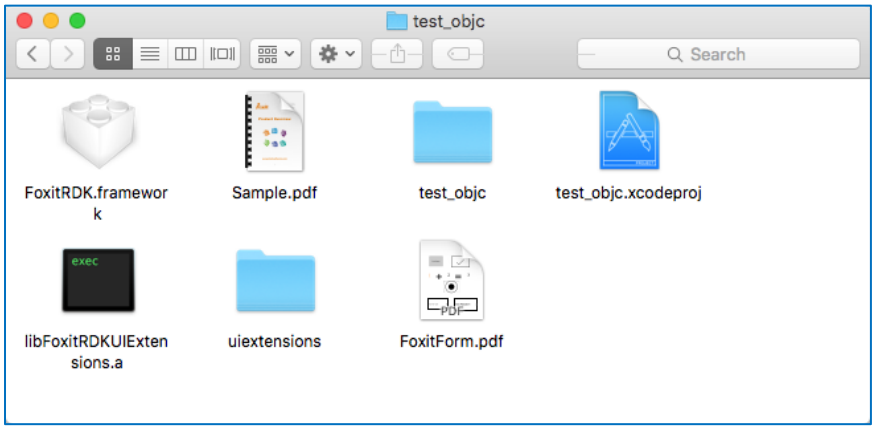


Figure 2-24

Step 4: Add the **Resource** files that are needed for the built-in UI implementations to the *test_objc* project. (Note: The Resource files might not be used for form filling feature, but required for the following sections.)

Right-click the *test_objc* project, and select **Add Files to "test_objc"**... to add the Resource files. Find and choose the folder as shown in Figure 2-25 .

Note *If you didn't copy the* ***uiextensions*** *file to "test_objc", please find and choose the file in the "libs/uiextensions_src/uiextensions" of the download package.*

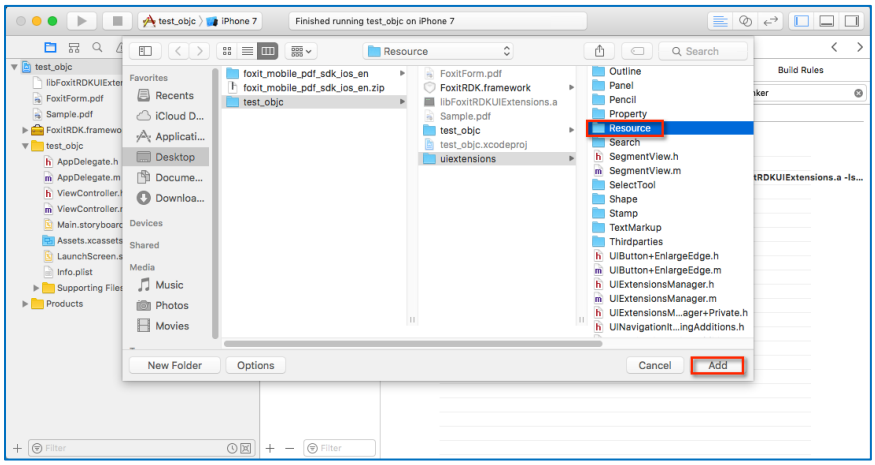


Figure 2-25

After completing the above four steps, the *test_objc* project will look like Figure 2-26.

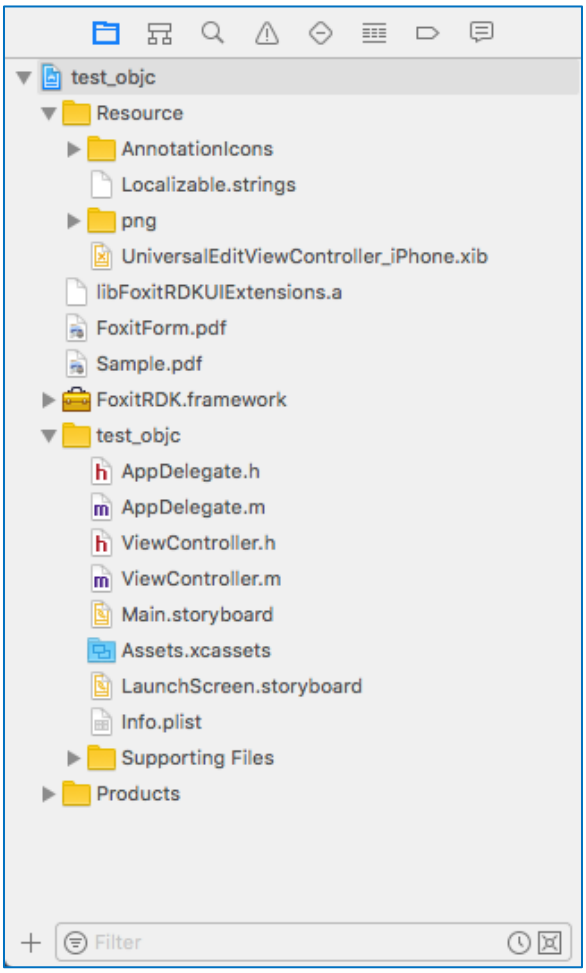


Figure 2-26

Add code to instantiate a UIExtensionsManager object and set it to PDFViewCtrl

In the "ViewController.m" file, you only need to add three lines of code to support form filling as follows:

Initialize a UIExtensionsManager object and set it to PDFViewCtrl.

```
#import "../uiextensions/UIExtensionsManager.h"

UIExtensionsManager* extensionsManager;
...
extensionsManager = [[UIExtensionsManager alloc] initWithPDFViewControl:pdfViewCtrl];
pdfViewCtrl.extensionsManager = extensionsManager;
```

The whole update of ViewController.m is as follows:

```
#import "ViewController.h"
#import "FoxitRDK/FSPDFViewControl.h"
#import "../uiextensions/UIExtensionsManager.h"
```

```
@interface ViewController ()


@end

@implementation ViewController
{
    UIExtensionsManager* extensionsManager;
}

- (void)viewDidLoad {
    [super viewDidLoad];

    // Get the path of a PDF
    NSString* pdfPath = [[NSBundle mainBundle] pathForResource:@"FoxitForm" ofType:@"pdf"];

    // Initialize a PDFDoc object with the path to the PDF file
    FSPDFDoc* pdfdoc = [FSPDFDoc createFromFilePath:pdfPath];
    if(e_errSuccess != [pdfdoc load:nil]) {
        return;
    }

    // Initialize a FSPDFViewCtrl object with the size of the entire screen
    FSPDFViewCtrl* pdfViewCtrl;
    pdfViewCtrl = [[FSPDFViewCtrl alloc] initWithFrame: [self.view bounds]];

    // Set the document to display
    [pdfViewCtrl setDoc:pdfdoc];

    // Add the pdfViewCtrl to the root view
    [self.view addSubview:pdfViewCtrl];


    // Initialize a UIExtensionsManager object and set it to pdfViewCtrl
    extensionsManager = [[UIExtensionsManager alloc] initWithPDFViewControl:pdfViewCtrl];
    pdfViewCtrl.extensionsManager = extensionsManager;
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.

}

@end
```

Now let's run it on an iPhone 7 Simulator. The "FoxitForm.pdf" will be displayed as shown in Figure 2-27. You can find that the Figure 2-27 is already different from the Figure 2-21. It means the form filling feature is available at present. Feel free to edit the form, such as Figure 2-28.

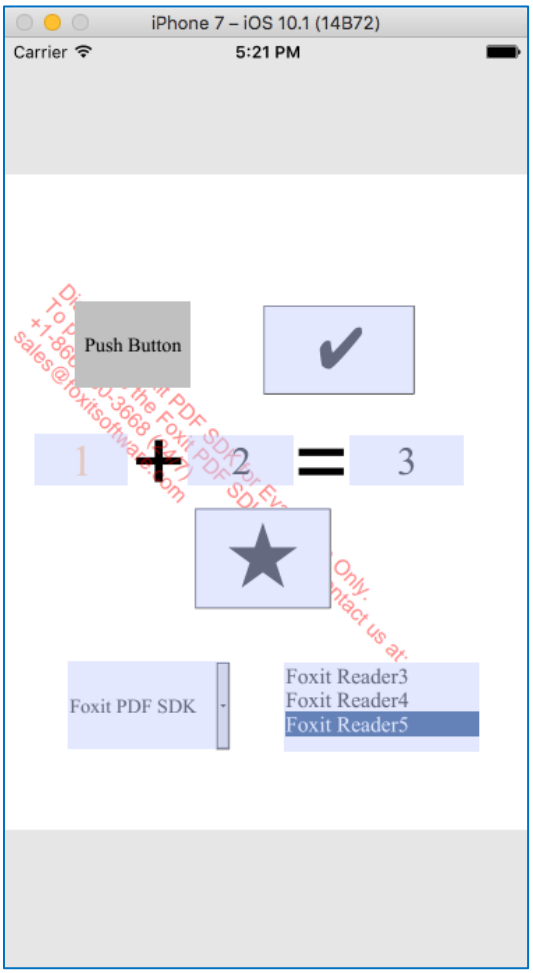

iPhone 7 – iOS 10.1 (14B72)
Carrier
5:21 PM
Push Button
1
2
3
Foxit PDF SDK
Foxit Reader3
Foxit Reader4
Foxit Reader5

Figure 2-27

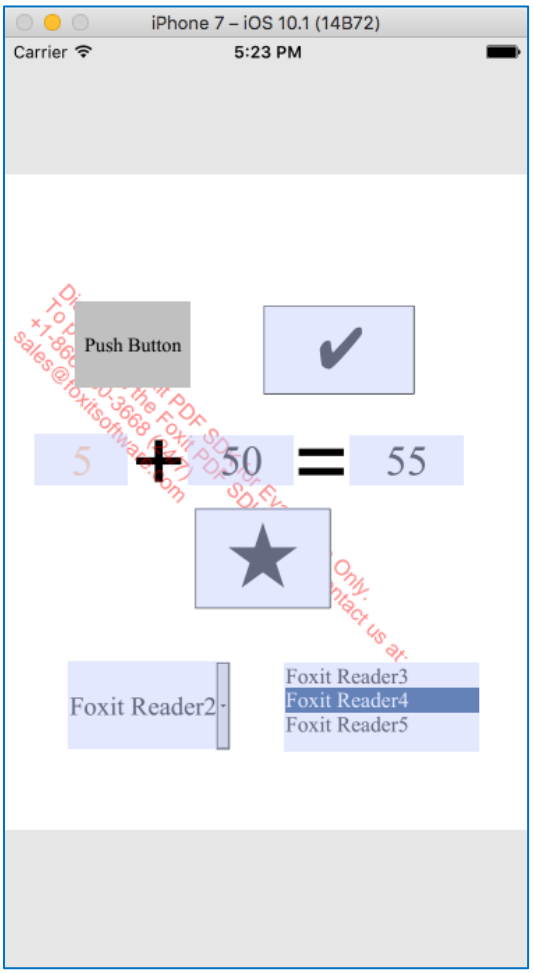


Figure 2-28

Amazing! We have realized the form filling feature in the *test_objc* project without adding any extra code related to forms. Alright, it is just a simple example which allows you to fill a PDF Form file, for further research about form, you can refer to the "complete_pdf_viewer" demo.

2.4.6 Add support for Text Search

Text search, annotations and related features are associated more closely with the user interface and as such require a slightly different approach from the form filling feature. We need to write some extra code to load the feature module and trigger the feature.

In this section, we will add support for text search and also extend the simple iOS app in the section "Displaying a PDF document". For annotations, you can refer to the "Add support for Annotations" section.

For simplicity, we will add a button on the main interface and use the button click event to quickly experience the text search feature. Just follow the steps below:

Step 1: Refer to "Preparatory work" and "Add code to instantiate a UIExtensionsManager object and set it to PDFViewCtrl" in the section "Add support for Form Filling" to add the same configuration and code in the *test_objc* project.

Step 2: In the "ViewController.m" file, we are now going to add the code necessary for triggering the search functionality. The required code additions are shown below and further down you will find a full example of what the "ViewController.m" file should look like.

Register the search event listener.

```
@interface ViewController () <ISearchEventListener>
...
[extensionsManager registerSearchEventListener:self];
```

Instantiate a Button object, add the click event, and set it to the root view.

```
UIButton* searchButton;
...
searchButton = [[UIButton alloc] initWithFrame:CGRectMake(280, 40, 80, 40)];
[searchButton setBackgroundColor:[UIColor grayColor]];
[searchButton setTitle: @"Search" forState: UIControlStateNormal];
[searchButton addTarget:self action:@selector(showSearchBar)
forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:searchButton];
```

The searchButton click event:

```
- (void)showSearchBar
{
    if (extensionsManager.currentAnnot) {

        [extensionsManager setCurrentAnnot:nil];
    }

    [extensionsManager showSearchBar:YES];
}
```

The search event listener:

```
- (void)onSearchStarted {
    searchButton.hidden = YES;
}

- (void)onSearchCanceled {
    searchButton.hidden = NO;
}
```

The whole update of ViewController.m is as follows:

```
#import "ViewController.h"
#import "FoxitRDK/FSPDFViewControl.h"
#import "../uiextensions/UIExtensionsManager.h"
```

```
@interface ViewController () <ISearchEventListener>


@end

@implementation ViewController
{
    UIExtensionsManager* extensionsManager;
    UIButton* searchButton;
}

- (void)viewDidLoad {
    [super viewDidLoad];

    // Get the path of a PDF
    NSString* pdfPath = [[NSBundle mainBundle] pathForResource:@"Sample" ofType:@"pdf"];

    // Initialize a PDFDoc object with the path to the PDF file
    FSPDFDoc* pdfdoc = [FSPDFDoc createFromFilePath:pdfPath];
    if(e_errSuccess != [pdfdoc load:nil]) {
        return;
    }

    // Initialize a FSPDFViewCtrl object with the size of the entire screen
    FSPDFViewCtrl* pdfViewCtrl;
    pdfViewCtrl = [[FSPDFViewCtrl alloc] initWithFrame: [self.view bounds]];

    // Set the document to display
    [pdfViewCtrl setDoc:pdfdoc];

    // Add the pdfViewCtrl to the root view
    [self.view addSubview:pdfViewCtrl];


    // Initialize a UIExtensionsManager object and set it to pdfViewCtrl
    extensionsManager = [[UIExtensionsManager alloc] initWithPDFViewControl:pdfViewCtrl];
    pdfViewCtrl.extensionsManager = extensionsManager;

    // Register the search event listener
    [extensionsManager registerSearchEventListener:self];


    // Instantiate a Button object and add the click event.
    searchButton = [[UIButton alloc] initWithFrame:CGRectMake(280, 40, 80, 40)];
    [searchButton setBackgroundColor:[UIColor grayColor]];
    [searchButton setTitle: @"Search" forState: UIControlStateNormal];
    [searchButton addTarget:self action:@selector(showSearchBar)
forControlEvents:UIControlEventTouchUpInside];

    // Add the searchButton to the root view
    [self.view addSubview:searchButton];
}

#pragma searchButton click event

- (void)showSearchBar
{
    if (extensionsManager.currentAnnot) {
```

```
        [extensionsManager setCurrentAnnot:nil];
    }

    [extensionsManager showSearchBar:YES];
}


#pragma ISearchEventListener

- (void)onSearchStarted {
    searchButton.hidden = YES;
}

- (void)onSearchCanceled {
    searchButton.hidden = NO;
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.

}

@end
```

Note *For this project, we use "Sample.pdf" as the test file. You can use any PDF file, just remember to add it to the project, and change the file path in "ViewController.m".*

Now that we have finished adding support for text search into the project, let's run it on an iPhone 7 Simulator. You will see that the "Sample.pdf" document is automatically displayed as shown in Figure 2-29.

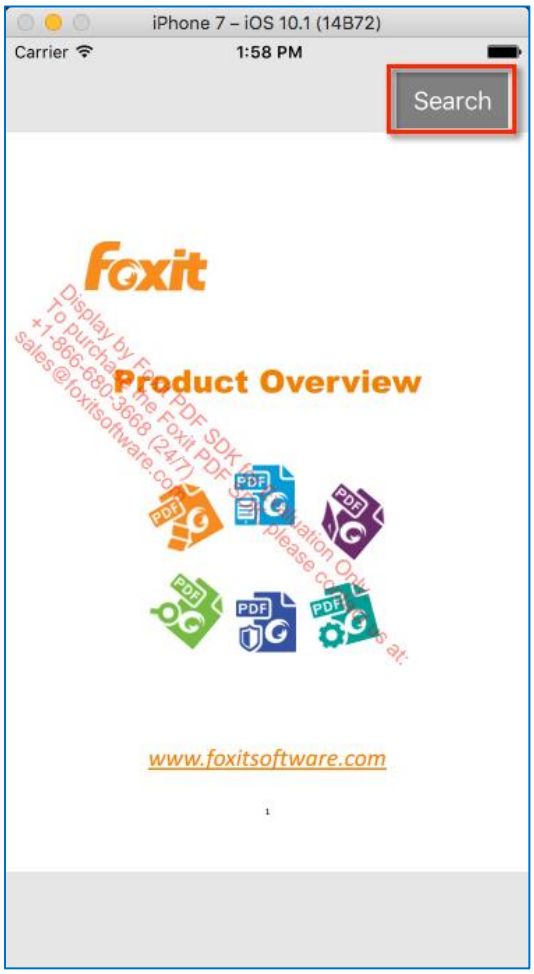


Figure 2-29

Click on the "**Search**" button, you can search anything you like. For example, input "Foxit", press "Enter", and then all of the search results will be listed as shown in Figure 2-30.

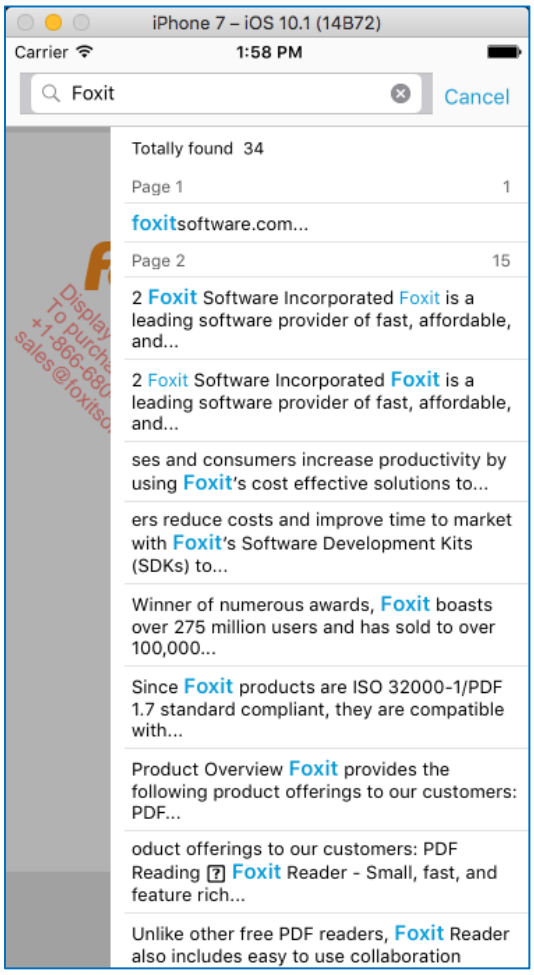


Figure 2-30

Click any result in the list to jump to the specific location. Here, we click the result in the second paragraph in the list, and then it will jump to the place where the result is, and the word will be highlighted as shown in Figure 2-31 (zoom in on the page to see it clearly). You can click the previous or next button to find the previous or next search result.

Figure 2-31

2.4.7 Add support for Annotations

Foxit MobilePDF SDK includes a wide variety of standard annotations, and each of them uses the similar way to be added into the project. In this section, we will show you how to add support for the common used annotations which are provided in the Complete PDF Viewer demo, and will take the highlight annotation as an example to set forth the steps in detail. For other common used annotations, we only list the core code, for more details, please refer to the highlight section.

2.4.7.1 Highlight

We will add the highlight feature to the simple iOS app in the section "Displaying a PDF document", and like previous added features, we also add a button on the main interface and use the button click event to quickly experience the highlight feature. Just follow the steps below:

Step 1: Refer to "Preparatory work" and "Add code to instantiate a UIExtensionsManager object and set it to PDFViewCtrl" in the section "Add support for Form Filling" to add the same configuration and code in the *test_objc* project.

Step 2: Left-click the *test_objc* project, navigate to the **Build Settings** tab, find the **Search Paths** section. Add the path of the **uiextensions** folder to **Header Search Paths** in the **Build Settings** tab as shown in Figure 2-32. (In this project, we assume that you have copied the **uiextensions** folder from the "libs/uiextensions_src" of the download package to "test_objc".)

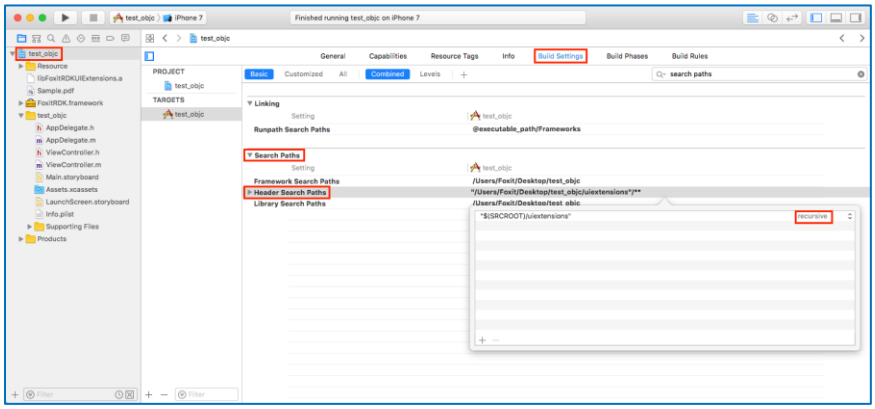


Figure 2-32

Step 3: In the "ViewController.m" file, we are now going to add the code necessary for triggering the highlight functionality. The required code additions are shown below and further down you will find a full example of what the "ViewController.m" file should look like.

Instantiate a Button object, add the click event, and set it to the root view.

```
UIButton* highlightButton;
...
highlightButton = [[UIButton alloc] initWithFrame:CGRectMake(280, 25, 80, 40)];
[highlightButton setBackgroundColor:[UIColor grayColor]];
[highlightButton setTitle:@"Highlight" forState:UIControlStateNormal];
[highlightButton addTarget:self action:@selector(highlightClick)
forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:highlightButton];
```

Handle the button click event. Get a MKToolHandler object from UIExtensionManager (the MKToolHandler is already instantiated in UIExtensionManager), set the type to highlight, and set it as the current tool handler.

```
#import "../uiextensions/TextMarkup/TextMKToolHandler.h"
...
```

```
MKToolHandler* mkToolHandler;
...

- (void)highlightClick
{
    mkToolHandler = [extensionsManager getToolHandlerByName:Tool_Markup];
    mkToolHandler.type = e_annotHighlight;
    [extensionsManager setCurrentToolHandler:mkToolHandler];
}
```

The whole update of ViewController.m is as follows:

```
#import "ViewController.h"
#import "FoxitRDK/FSPDFViewControl.h"
#import "../uiextensions/UIExtensionsManager.h"
#import "../uiextensions/TextMarkup/TextMKToolHandler.h"

@interface ViewController : UIViewController

@end

@implementation ViewController
{
    FSPDFViewCtrl* pdfViewCtrl;
    UIExtensionsManager* extensionsManager;
    UIButton* highlightButton;
    MKToolHandler* mkToolHandler;
}

- (void)viewDidLoad {
    [super viewDidLoad];

    // Get the path of a PDF.
    NSString* pdfPath = [[NSBundle mainBundle] pathForResource:@"Sample" ofType:@"pdf"];

    // Initialize a PDFDoc object with the path to the PDF file.
    FSPDFDoc* pdfdoc = [FSPDFDoc createFromFilePath:pdfPath];
    if(e_errSuccess != [pdfdoc load:nil]) {
        return;
    }

    // Initilize a FSPDFViewCtrl object with the size of the entire screen.
    pdfViewCtrl = [[FSPDFViewCtrl alloc] initWithFrame: [self.view bounds]];

    // Set the document to display
    [pdfViewCtrl setDoc:pdfdoc];

    // Add the pdfViewCtrl to the root view.
```

```
    [self.view addSubview:pdfViewCtrl];

    // Initialize a UIExtensionsManager object and set it to pdfViewCtrl.
    extensionsManager = [[UIExtensionsManager alloc] initWithPDFViewControl:pdfViewCtrl];
    pdfViewCtrl.extensionsManager = extensionsManager;

    // Instantiate a Button object and add the click event.
    highlightButton = [[UIButton alloc] initWithFrame:CGRectMake(280, 25, 80, 40)];
    [highlightButton setBackgroundColor:[UIColor grayColor]];
    [highlightButton setTitle:@"Highlight" forState:UIControlStateNormal];
    [highlightButton addTarget:self action:@selector(highlightClick)
forControlEvents:UIControlEventTouchUpInside];

    // Add the highlightButton to the root view.
    [self.view addSubview:highlightButton];
}

#pragma HighlightButton click event
- (void)highlightClick
{
    mkToolHandler = [extensionsManager getToolHandlerByName:Tool_Markup];
    mkToolHandler.type = e_annotHighlight;
    [extensionsManager setCurrentToolHandler:mkToolHandler];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

Note *For this project, we use "Sample.pdf" as the test file. You can use any PDF file, just remember to add it to the project, and change the file path in "ViewController.m".*

Now that we have finished adding support for highlight into the project, let's run it on an iPhone 7 Simulator. You will see that the "Sample.pdf" document is automatically displayed (see Figure 2-33).

Figure 2-33

In order to see the highlight result clearly, swipe right to the next page, click the "**Highlight**" button, and drag over text to highlight the selected text (see Figure 2-34).

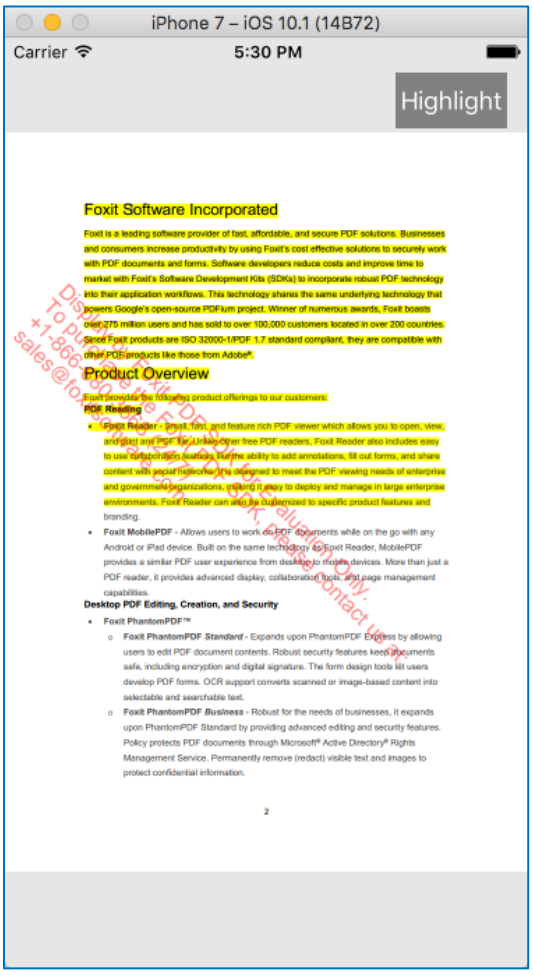


Figure 2-34

2.4.7.2 Underline

Adding support for underline is similar to add support for highlight. The core point is to get a MKToolHandler object from UIExtensionManager (the MKToolHandler is already instantiated in UIExtensionManager), set the type to underline, and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for underline. The core code is below:

```
#import "../uiextensions/TextMarkup/TextMKToolHandler.h"
...

MKToolHandler* mkToolHandler;
...

- (void)underlineClick
{
    mkToolHandler = [extensionsManager getToolHandlerByName:Tool_Markup];
    mkToolHandler.type = e_annotUnderline;
```

```
    [extensionsManager setCurrentToolHandler:mkToolHandler];
}
```

2.4.7.3 Squiggly

Adding support for squiggly is similar to add support for highlight. The core point is to get a MKToolHandler object from UIExtensionManager (the MKToolHandler is already instantiated in UIExtensionManager), set the type to squiggly, and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for squiggly. The core code is below:

```
#import "../uiextensions/TextMarkup/TextMKToolHandler.h"
...

MKToolHandler* mkToolHandler;
...

- (void)squigglyClick
{
    mkToolHandler = [extensionsManager getToolHandlerByName:Tool_Markup];
    mkToolHandler.type = e_annotSquiggly;
    [extensionsManager setCurrentToolHandler:mkToolHandler];
}
```

2.4.7.4 Strikeout

Adding support for strikeout is similar to add support for highlight. The core point is to get a MKToolHandler object from UIExtensionManager (the MKToolHandler is already instantiated in UIExtensionManager), set the type to strikeout, and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for strikeout. The core code is below:

```
#import "../uiextensions/TextMarkup/TextMKToolHandler.h"
...

MKToolHandler* mkToolHandler;
...

- (void)strikeoutClick
{
    mkToolHandler = [extensionsManager getToolHandlerByName:Tool_Markup];
    mkToolHandler.type = e_annotStrikeOut;
    [extensionsManager setCurrentToolHandler:mkToolHandler];
}
```

2.4.7.5 Insert text

Adding support for insert text is similar to add support for highlight. The core point is to get an InsertToolHandler object from UIExtensionManager (the InsertToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for insert text. The core code is below:

```
#import "../uiextensions/Caret/InsertToolHandler.h"
...

InsertToolHandler* insertToolHandler;
...

- (void)insertClick
{
    insertToolHandler = [extensionsManager getToolHandlerByName:Tool_Insert];
    [extensionsManager setCurrentToolHandler:insertToolHandler];
}
```

2.4.7.6 Replace text

Adding support for replace text is similar to add support for highlight. The core point is to get a ReplaceToolHandler object from UIExtensionManager (the ReplaceToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for replace text. The core code is below:

```
#import "../uiextensions/Caret/ReplaceToolHandler.h"
...

ReplaceToolHandler* replaceToolHandler;
...

- (void)replaceClick
{
    replaceToolHandler = [extensionsManager getToolHandlerByName:Tool_Replace];
    [extensionsManager setCurrentToolHandler:replaceToolHandler];
}
```

2.4.7.7 Line

Line and arrow tools are implemented in LineToolHandler. So, the core point to add support for line is to get a LineToolHandler object from UIExtensionManager (the LineToolHandler is already instantiated in UIExtensionManager), set the type to line (default type is also line), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for line. The core code is below:

```
#import "../uiextensions/Line/LineToolHandler.h"
```

```
...

LineToolHandler* lineToolHandler;
...

- (void)lineClick
{
    lineToolHandler = [extensionsManager getToolHandlerByName:Tool_Line];
    lineToolHandler.type = e_annotLine; // default type is also line.
    [extensionsManager setCurrentToolHandler:lineToolHandler];
}
```

2.4.7.8 Arrow

Line and arrow tools are implemented in LineToolHandler. So, the core point to add support for arrow is to get a LineToolHandler object from UIExtensionManager (the LineToolHandler is already instantiated in UIExtensionManager), set the "isArrowLine" to "YES", and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for arrow. The core code is below:

```
#import "../uiextensions/Line/LineToolHandler.h"
...

LineToolHandler* lineToolHandler;
...

- (void)arrowClick
{
    lineToolHandler = [extensionsManager getToolHandlerByName:Tool_Line];
    lineToolHandler.isArrowLine = YES;
    [extensionsManager setCurrentToolHandler:lineToolHandler];
}
```

2.4.7.9 Square

Square and circle tools are implemented in ShapeToolHandler. So, the core point to add support for square is to get a ShapeToolHandler object from UIExtensionManager (the ShapeToolHandler is already instantiated in UIExtensionManager), set the type to square, and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for square. The core code is below:

```
#import "../uiextensions/Line/ShapeToolHandler.h"
...

ShapeToolHandler* shapeToolHandler;
...

- (void)squareClick
{
    shapeToolHandler = [extensionsManager getToolHandlerByName:Tool_Shape];
```

```
    shapeToolHandler.type = e_annotSquare;
    [extensionsManager setCurrentToolHandler:shapeToolHandler];
}
```

2.4.7.10 Circle

Square and circle tools are implemented in ShapeToolHandler. So, the core point to add support for circle is to get a ShapeToolHandler object from UIExtensionManager (the ShapeToolHandler is already instantiated in UIExtensionManager), set the type to circle, and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for circle. The core code is below:

```
#import "../uiextensions/Line/ShapeToolHandler.h"
...

ShapeToolHandler* shapeToolHandler;
...

- (void)circleClick
{
    shapeToolHandler = [extensionsManager getToolHandlerByName:Tool_Shape];
    shapeToolHandler.type = e_annotCircle;
    [extensionsManager setCurrentToolHandler:shapeToolHandler];
}
```

2.4.7.11 Pencil

Adding support for pencil is similar to add support for highlight. The core point is to get a PencilToolHandler object from UIExtensionManager (the PencilToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for pencil. The core code is below:

```
#import "../uiextensions/Pencil/Pencil/PencilToolHandler.h"
...

PencilToolHandler* pencilToolHandler;
...

- (void)pencilClick
{
    pencilToolHandler = (PencilToolHandler*)[extensionsManager
getToolHandlerByName:Tool_Pencil];
    [extensionsManager setCurrentToolHandler:pencilToolHandler];
}
```

2.4.7.12 Eraser

Eraser tool is generally used in combination with the pencil tool. The core point to add support for eraser is to get an EraseToolHandler object from UIExtensionManager (the EraseToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for eraser. The core code is below:

```
#import "../uiextensions/Pencil/Erase/EraseToolHandler.h"
...

EraseToolHandler* eraseToolHandler;
...

- (void)eraserClick
{
    eraseToolHandler = [extensionsManager getToolHandlerByName:Tool_Eraser];
    [extensionsManager setCurrentToolHandler: eraseToolHandler];
}
```

2.4.7.13 Typewriter

Adding support for typewriter is similar to add support for highlight. The core point is to get a FtToolHandler object from UIExtensionManager (the FtToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for typewriter. The core code is below:

```
#import "../uiextensions/Freetext/FtToolHandler.h"
...

FtToolHandler* ftToolHandler;
...

- (void)typewriterClick
{
    ftToolHandler = (FtToolHandler*)[extensionsManager getToolHandlerByName:Tool_Freetext];
    [extensionsManager setCurrentToolHandler: ftToolHandler];
}
```

2.4.7.14 Note

Adding support for note is similar to add support for highlight. The core point is to get a NoteToolHandler object from UIExtensionManager (the NoteToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for note. The core code is below:

```
#import "../uiextensions/Note/NoteToolHandler.h"
```

```
...

NoteToolHandler* noteToolHandler;
...

- (void)noteClick
{
    noteToolHandler = [extensionsManager getToolHandlerByName:Tool_Note];
    [extensionsManager setCurrentToolHandler: noteToolHandler];
}
```

2.4.7.15 Stamp

Adding support for stamp is similar to add support for highlight. The core point is to get a StampToolHandler object from UIExtensionManager (the StampToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for stamp. The core code is below:

```
#import "../uiextensions/Stamp/StampToolHandler.h"
...

StampToolHandler* stampToolHandler;
...

- (void)stampClick
{
    stampToolHandler = [extensionsManager getToolHandlerByName:Tool_Stamp];
    [extensionsManager setCurrentToolHandler: stampToolHandler];
}
```

2.4.7.16 Attachment

Adding support for attachment is similar to add support for highlight. The core point is to get an AttachmentToolHandler object from UIExtensionManager (the AttachmentToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for attachment. The core code is below:

```
#import "../uiextensions/Attachment/AttachmentToolHandler.h"
...

AttachmentToolHandler* attachmentToolHandler;
...

- (void)attachmentClick
{
    attachmentToolHandler = [extensionsManager getToolHandlerByName:Tool_Attachment];
    [extensionsManager setCurrentToolHandler: attachmentToolHandler];
}
```

2.5 How to make an iOS app in Swift with Foxit MobilePDF SDK

Nowadays, Swift is more and more popular for iOS developers because its syntax is much cleaner and easier to read. To better support Swift developers, this section will help you to quickly get started with using Foxit MobilePDF SDK to make an iOS app in Swift with step-by-step instructions provided. From now, you can get familiar with Foxit MobilePDF SDK and use Swift to create your first PDF iOS app in Xcode. This section includes the following steps:

1) Create a new iOS project in Swift
2) Integrate Foxit MobilePDF SDK into your apps
3) Apply the license key
4) Display a PDF document
5) Add support for Form Filling
6) Add support for Text Search
7) Add support for Annotations

2.5.1 Create a new iOS project in Swift

In this guide, we use Xcode 8.1 to create a new iOS project.

Fire up Xcode, choose **File** -> **New** -> **Project**..., and then select **iOS** -> **Single View Application** as shown in Figure 2-35. Click **Next**.

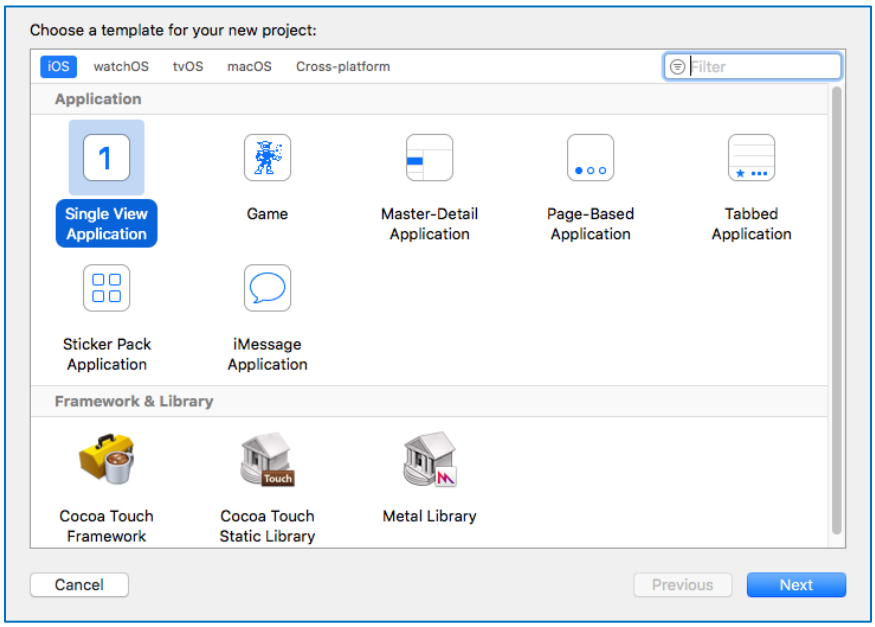


Figure 2-35

Choose the options for your new project as shown in Figure 2-36. Please make sure to choose Swift as the programming language. For simplicity, we don't check the Unit Tests and UI Tests which are used for automated testing. Then, Click **Next**.

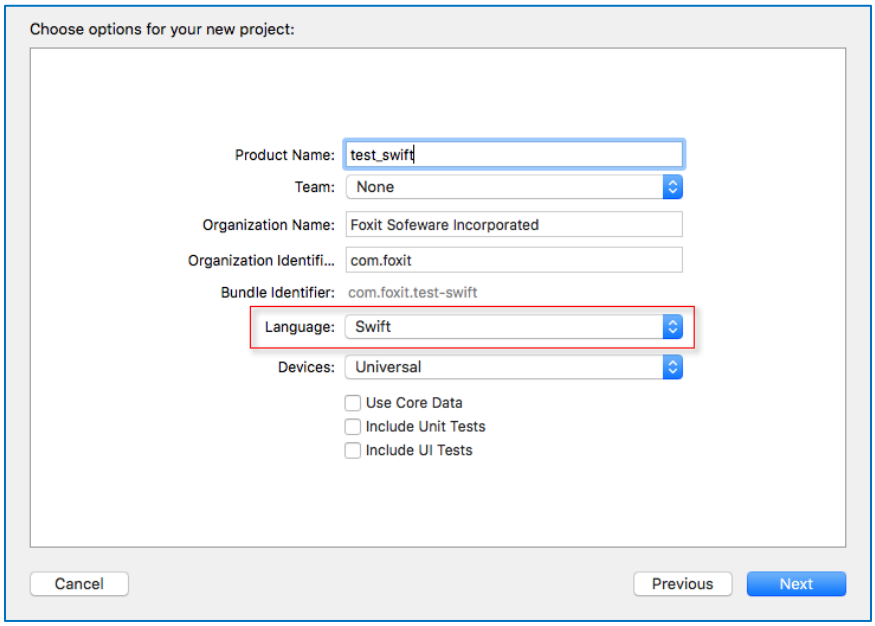


Figure 2-36

Place the project to the location as desired. The option "version control" is not actually important for building your first PDF app, so let's use the default setting. Here, we place the project to the desktop as shown in Figure 2-37. Then, click **Create**.

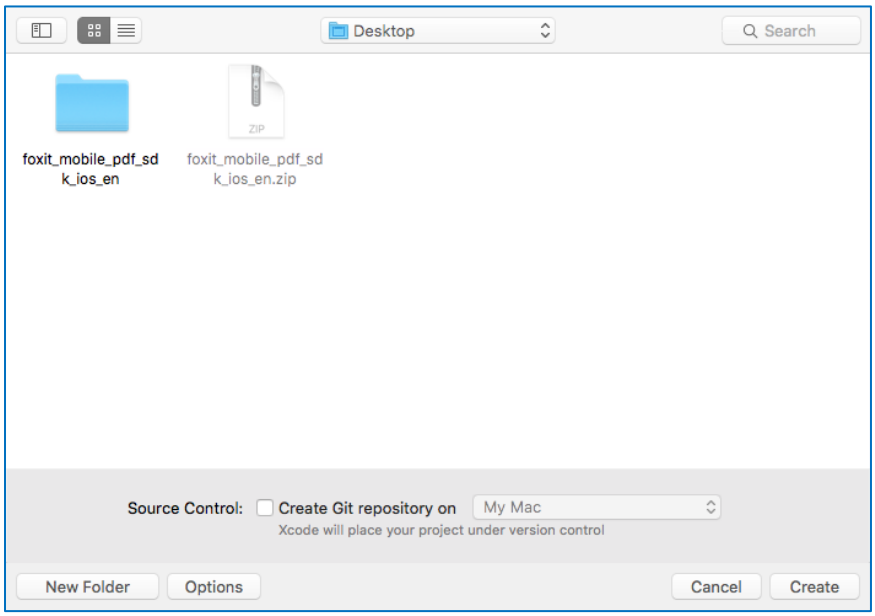


Figure 2-37

2.5.2 Integrate Foxit MobilePDF SDK into your apps

To integrate Foxit MobilePDF SDK into your apps, please first refer to section 2.4.2 "Integrate Foxit MobilePDF SDK into your apps" to add the dynamic framework "**FoxitRDK.framework**" into the *test_swift* project. Then, create a Swift bridging header which is used for building a bridge between Swift and Objective-C. That means it allows you to communicate with the Objective-C classes from your Swift classes. To create and configure Swift bridging header, please follow the steps blew:

a) In *test_swift* project, choose **File** -> **New** -> **File**..., and then select **iOS** -> **Header File** as shown in Figure 2-38. Click **Next**.

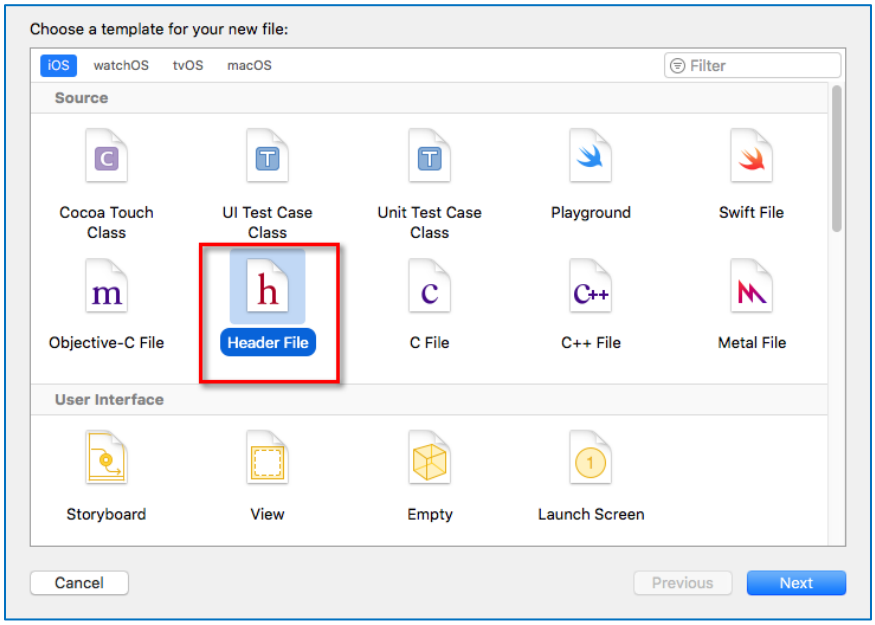


Figure 2-38

b) Name the header file. Example: in this project, the file is named "Bridging-Header" as shown in Figure 2-39. Click **Create**. Then the *test_swift* project will look like the Figure 2-40.

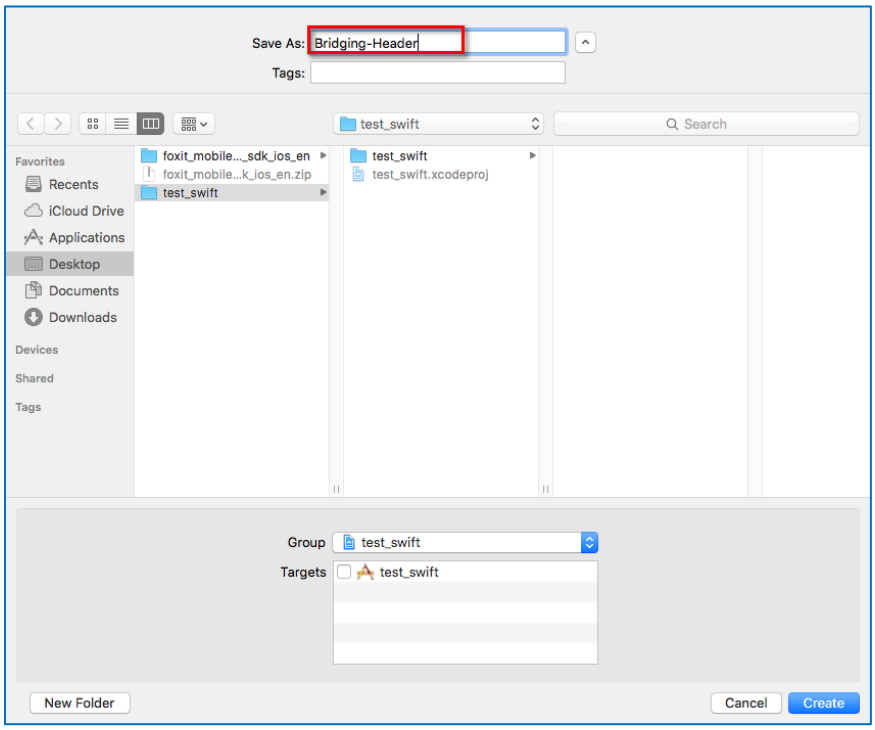


Figure 2-39

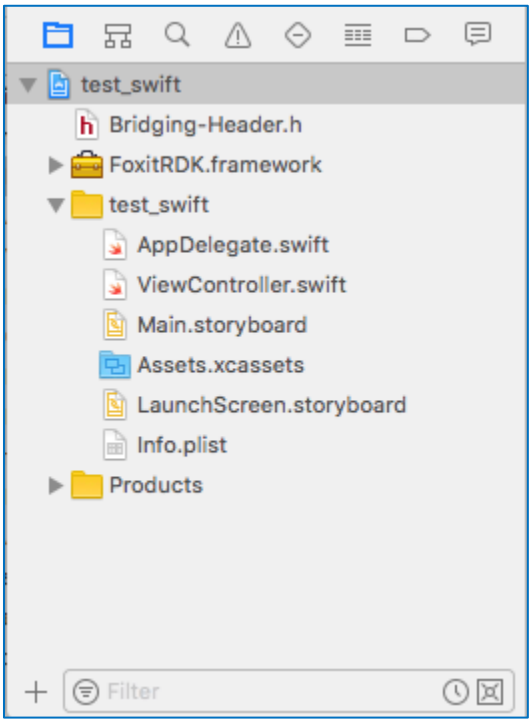


Figure 2-40

c) Left-click the *test_swift* project, navigate to the **Build Settings** tab, find the **Swift Compiler – General** section. You may find it faster to type in "Swift Compiler" into the search box to narrow down the results.

Next to **Objective-C Bridging Header**, add the created Swift bridging header "Bridging-Header.h" as shown in Figure 2-41.

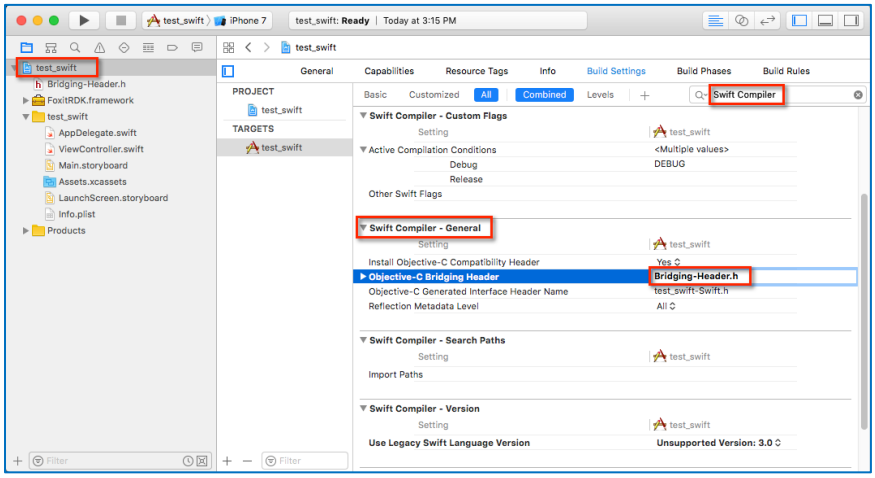


Figure 2-41

d) Open the "Bridging-Header.h" file, import the Objective-C classes you want to call using #import statements. Any class listed in this file will be able to be accessed from your Swift classes. Now, we import the following two classes first which are required for the section "Display a PDF document". (See Figure 2-42)

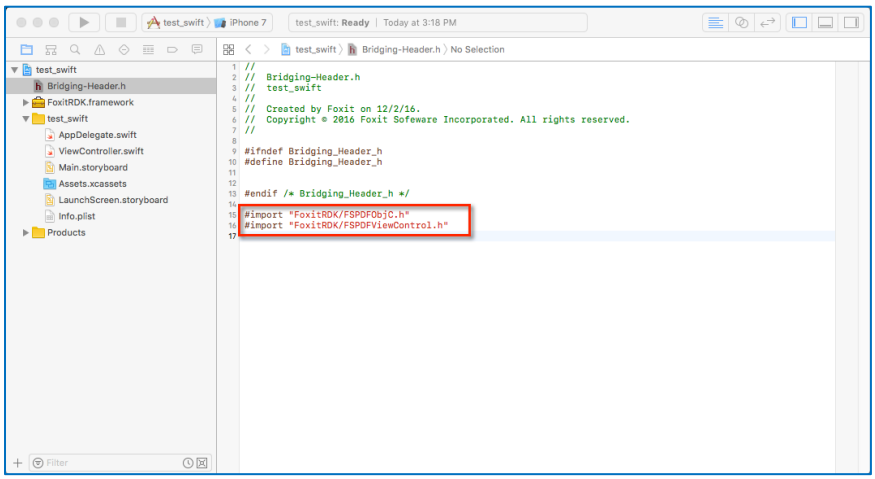


Figure 2-42

Now, we have created and configured a Swift bridging header in the *test_swift* project successfully.

2.5.3 Apply the license key

It is necessary for apps to initialize and unlock Foxit MobilePDF SDK using a license before calling any APIs. The function *FSLibrary.init(sn key:key)* is provided to initialize Foxit MobilePDF SDK. The trial license files can be found in the "libs" folder of the download package. After the evaluation period expires, you should purchase an official license to continue using it. Finish the initialization in the application method within the AppDelegate.swift file.

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {

        let sn = ""
        let key = ""
        let eRet = FSLibrary.init(sn, key:key)
        if e_errSuccess != eRet {
            return false
        }
        return true
}
```

Note *The parameter "sn" can be found in the "**rdk_sn.txt**" (the string after "SN=") and the "key" can be found in the "**rdk_key.txt**" (the string after "Sign=").*

2.5.4 Display a PDF document

So far, we have added "***FoxitRDK.framework***" to the *test_swift* project, and finished the initialization of the Foxit MobilePDF SDK. Now, let's start building a simple PDF viewer with just a few lines of code.

Note: *The UI Extensions Component is not required if you only need to display a PDF document.*

First of all, add a PDF file to the project which will be used as the test file. For example, we use "Sample.pdf" found in the "samples\test_files" folder of the download package. Right-click the *test_swift* project, and select **Add Files to "test_swift"**… to add this file. After adding, you can see the PDF in the Xcode's **Copy Bundle Resources** as shown in Figure 2-43.

Note *You can add the PDF to **Copy Bundle Resources** directly. Just left-click the test_swift project, find **Copy Bundle Resources** in the **Build Phases** tab, press on the **+** button, and choose the file to add. You can refer to any PDF file, just add it to the Xcode's **Copy Bundle Resources**.*

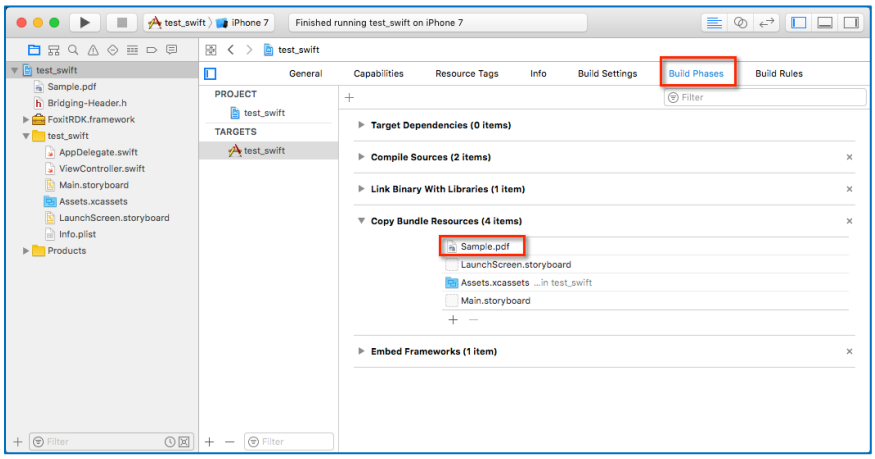


Figure 2-43

Then, add the following code to ViewController.swift to display a PDF document. It's really easy to present a PDF on screen. All you need is to create a **FSPDFDoc** object and then show it with a **FSPDFViewCtrl** object.

Update ViewController.swift as follows:

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        // Get the path of a PDF.
        let pdfPath = Bundle.main.path(forResource: "Sample", ofType: "pdf")!

        // Initialize a PDFDoc object with the path to the PDF file.
        let doc = FSPDFDoc.create(fromFilePath:pdfPath)
        if e_errSuccess != doc?.load(nil) {
            return
        }

        // Initialize a FSPDFViewCtrl object with the size of the entire screen.
        var pdfViewCtrl: FSPDFViewCtrl!
        pdfViewCtrl = FSPDFViewCtrl(frame:self.view.bounds)

        // Set the document to display.
        pdfViewCtrl.setDoc(doc)

        // Add the pdfViewCtrl to the root view.
        self.view.insertSubview(pdfViewCtrl, at: 0)
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

Fantastic! We have now finished building a simple iOS app in Swift which uses Foxit MobilePDF SDK to display a PDF document with just a few lines of code. The next step is to run the project on a device or simulator.

In this guide, we build and run the project on an iPhone 7 Simulator, and you will see that the "Sample.pdf" document is displayed as shown in Figure 2-44. Now, this sample app has some basic PDF features, such as zooming in/out and page turning. Just have a try!

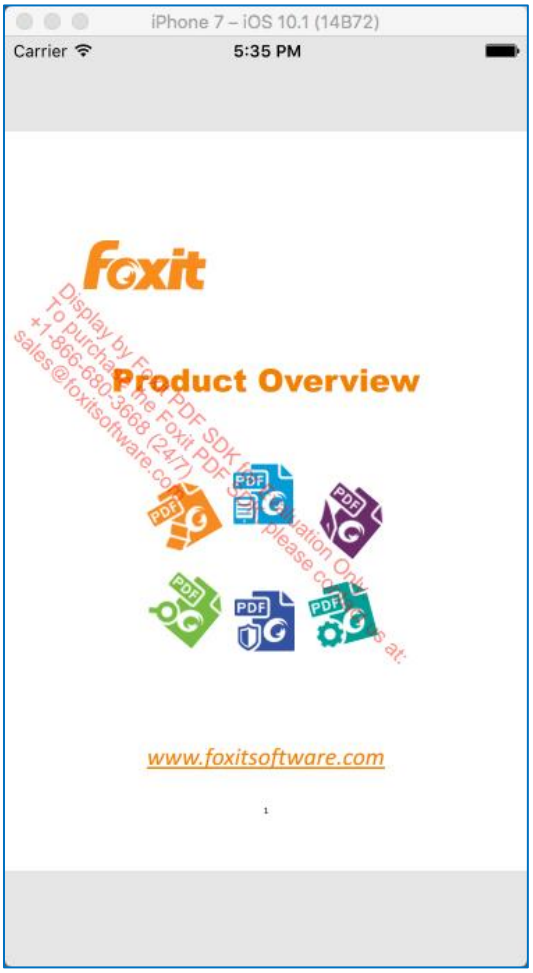


Figure 2-44

2.5.5 Add support for Form Filling

Foxit MobilePDF SDK comes with built-in support for features such as annotations, text search, outline and form filling. These visual features are implemented using Foxit MobilePDF SDK API and are shipped in the UI Extensions Component.

The form filling feature is already provided in the UI Extensions Component. It's simple and easy to integrate it into your app. For annotations and text search support, you can refer to the following sections.

In the previous sections, we have introduced how to add Foxit MobilePDF SDK to a project and how to build a simple iOS app for displaying a PDF document in Swift. Now, let's extend the simple app (*test_swift*) further to learn how to add support for form filling.

First, let's do a test. Prepare a PDF form file, and add it to the *test_swift* project. For example, we use a PDF form file called "FoxitForm.pdf" found in the "samples/test_files" folder of the download package.

Open "ViewController.swift", only change the file name from "Sample.pdf" to "FoxitForm.pdf" as follows:

```
NSString* pdfPath = [[NSBundle mainBundle] pathForResource:@"FoxitForm" ofType:@"pdf"];
```

Then rebuild and run the project, and you will see that the "FoxitForm.pdf" is displayed as shown in Figure 2-45. Now, it is just like a normal PDF file, in which the form fields cannot be edited.

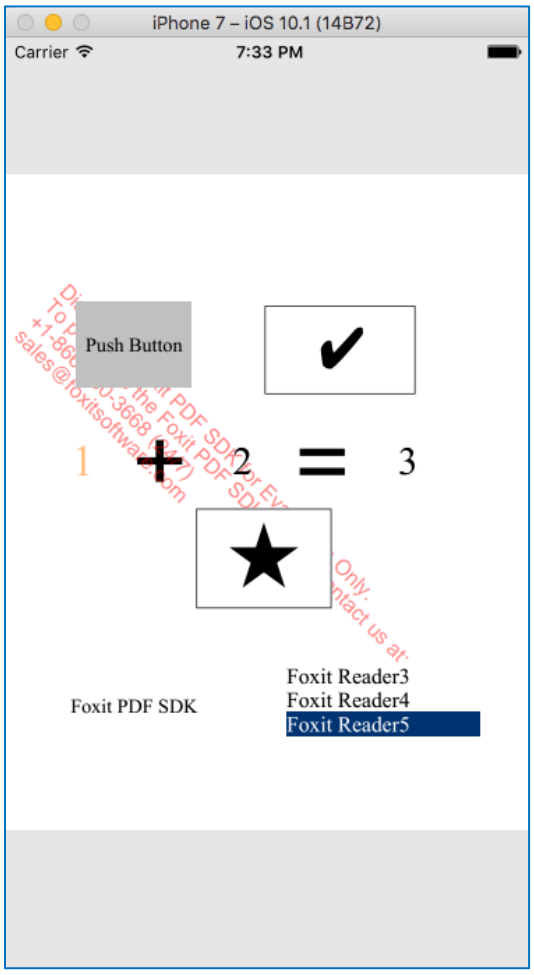


Figure 2-45

Next, let's add support for form filling. It's extremely easy! The key point is to instantiate a UIExtensionsManager object and set it to PDFViewCtrl. Please do the following preparatory work first, and then add code to instantiate a UIExtensionsManager object and set it to PDFViewCtrl.

Preparatory work

Step 1: Add UI Extensions Component (***libFoxitRDKUIExtensions.a***) to the project.

Note *In this app, we use the default built-in UI implementations to develop it, for simplicity and convenience, we will add* ***libFoxitRDKUIExtensions.a*** *to the test_swift project.*

Right-click the *test_swift* project, and select **Add Files to "test_swift"**… to add the extensions library. After adding, you can see the library in the Xcode's **Link Binary With Libraries** as shown in Figure 2-46.

Note *You can add the library to* ***Link Binary With Libraries*** *directly. Just left-click the test_swift project, find* ***Link Binary With Libraries*** *in the* ***Build Phases*** *tab, press on the* ***+*** *button, and choose the file to add. In either case, please check the "****Copy items if needed****" option when choosing the file.*

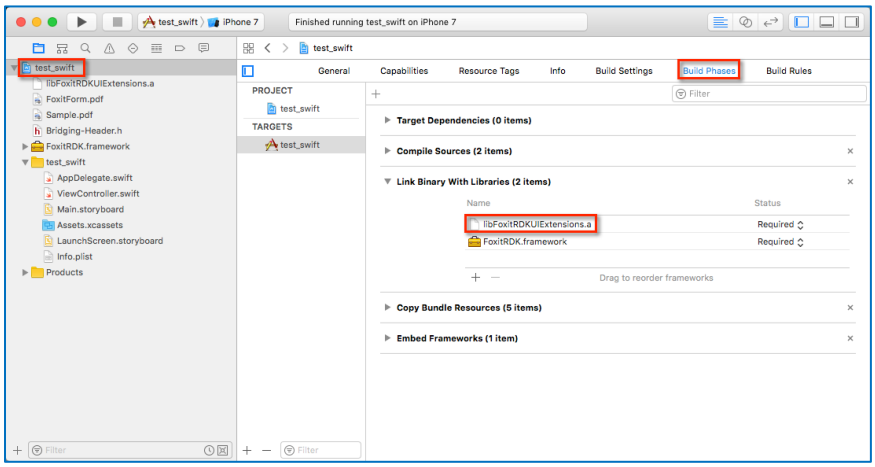


Figure 2-46

Step 2: Add "***-force_load libFoxitRDKUIExtensions.a -lstdc++***" to **Other Linker Flags** in the **Build Settings** tab as shown in Figure 2-47. "***-force_load libFoxitRDKUIExtensions.a"*** is used to load all of the members that implement any Objective-C class or category in the static library. "***-lstdc++***" ensures that the C++ standard library that is required by Foxit MobilePDF SDK will be included at link time.

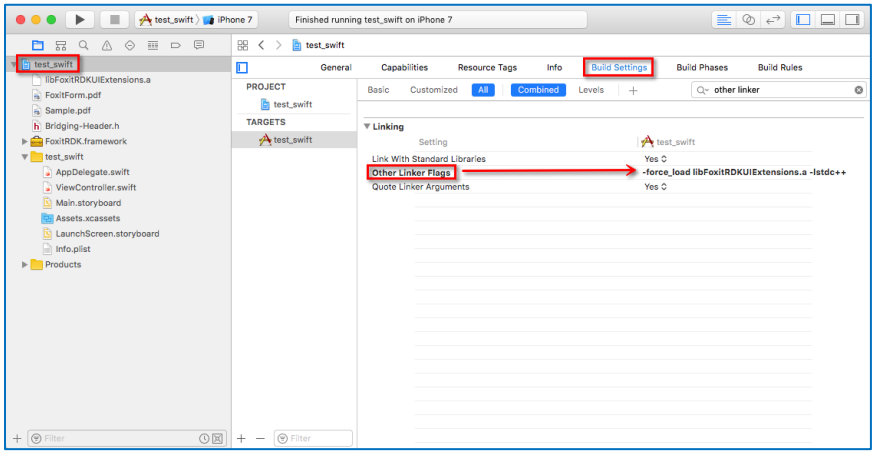


Figure 2-47

Step 3: Copy the **uiextensions** folder from the "libs/uiextensions_src" of the download package to "test_swift". This file contains the header files for ***libFoxitRDKUIExtensions.a***. Then, the *"test_swift"* folder will look like Figure 2-48.

Note *This project only needs the "UIExtensionsManager.h" file. So you can just add this header file found in the "libs/uiextensions_src/uiextensions" of the download package to the project.*

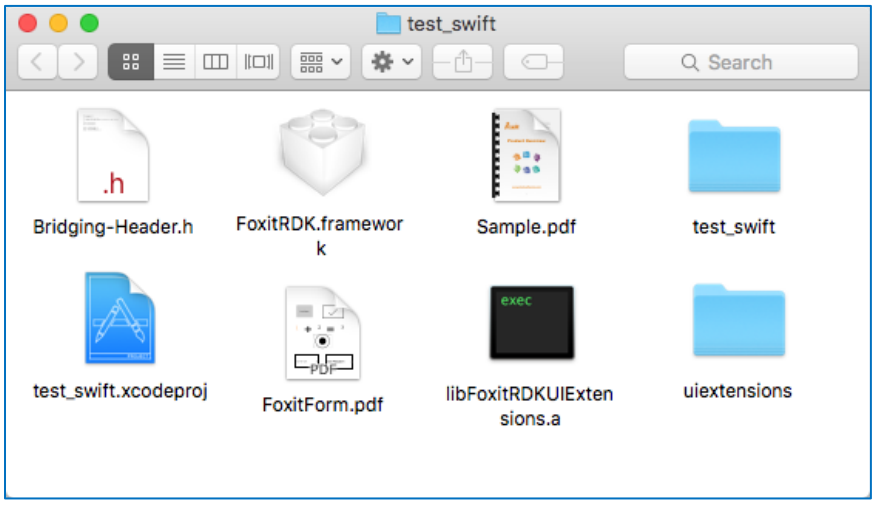


Figure 2-48

Step 4: Add the **Resource** files that are needed for the built-in UI implementations to the *test_swift* project. (Note: The Resource files might not be used for form filling feature, but required for the following sections.)

Right-click the *test_swift* project, and select **Add Files to "test_swift"**... to add the Resource files. Find and choose the folder as shown in Figure 2-49.

Note *If you didn't copy the* ***uiextensions*** *file to "test_swift", please find and choose the file in the "libs/uiextensions_src/uiextensions" of the download package.*

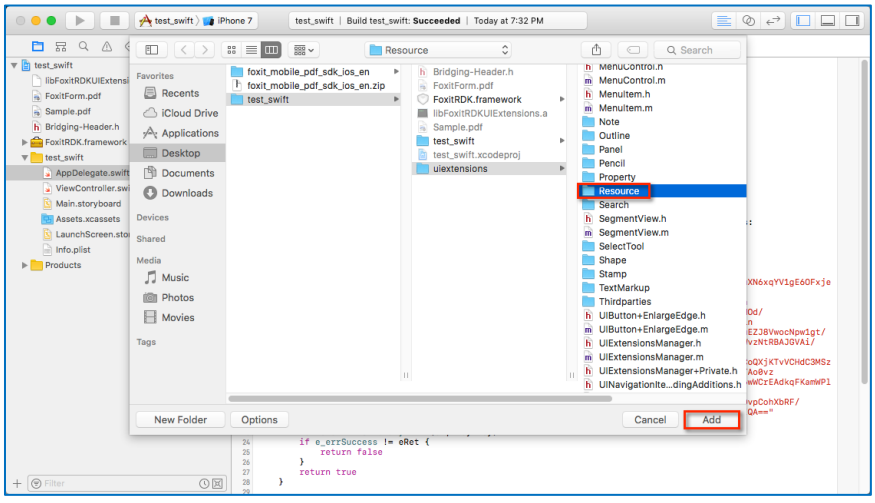


Figure 2-49

After completing the above four steps, the *test_swift* project will look like Figure 2-50.

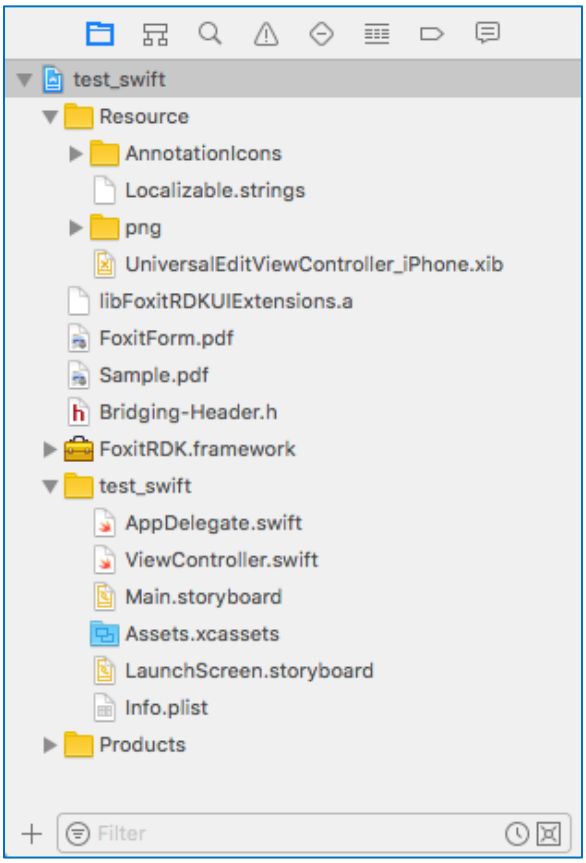


Figure 2-50

Add code to instantiate a UIExtensionsManager object and set it to PDFViewCtrl

First, open the "Bridging-Header.h" file, import the following header file. (See Figure 2-51)

```
#import "uiextensions/UIExtensionsManager.h"
```

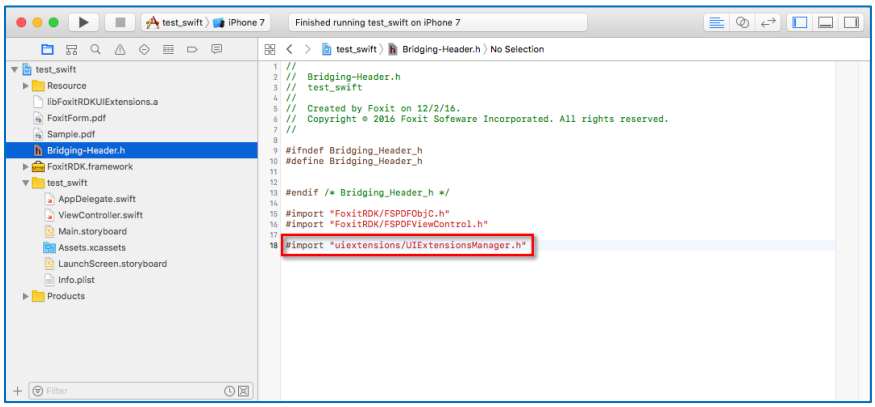


Figure 2-51

Then, in the "ViewController.swift" file, you only need to add three lines of code to support form filling as follows:

Initialize a UIExtensionsManager object and set it to PDFViewCtrl.

```
var extensionsManager: UIExtensionsManager!
...
extensionsManager = UIExtensionsManager(pdfViewControl: pdfViewCtrl)
pdfViewCtrl.extensionsManager = extensionsManager;
```

The whole update of ViewController.swift is as follows:

```
import UIKit


class ViewController: UIViewController {

    var extensionsManager: UIExtensionsManager!

    override func viewDidLoad() {
        super.viewDidLoad()

        // Get the path of a PDF.
        let pdfPath = Bundle.main.path(forResource: "FoxitForm", ofType: "pdf")!

        // Initialize a PDFDoc object with the path to the PDF file.
        let doc = FSPDFDoc.create(fromFilePath:pdfPath)
        if e_errSuccess != doc?.load(nil) {
            return
        }

        // Initialize a FSPDFViewCtrl object with the size of the entire screen.
        var pdfViewCtrl: FSPDFViewCtrl!
        pdfViewCtrl = FSPDFViewCtrl(frame:self.view.bounds)

        // Set the document to display.
        pdfViewCtrl.setDoc(doc)

        // Add the pdfViewCtrl to the root view.
```

```
        self.view.insertSubview(pdfViewCtrl, at: 0)

        // Initialize a UIExtensionsManager object and set it to pdfViewCtrl.
        extensionsManager = UIExtensionsManager(pdfViewControl: pdfViewCtrl)
        pdfViewCtrl.extensionsManager = extensionsManager;

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

Now let's run it on an iPhone 7 Simulator. The "FoxitForm.pdf" will be displayed as shown in Figure 2-52. You can find that the Figure 2-52 is already different from the Figure 2-45. It means the form filling feature is available at present. Feel free to edit the form, such as Figure 2-53.

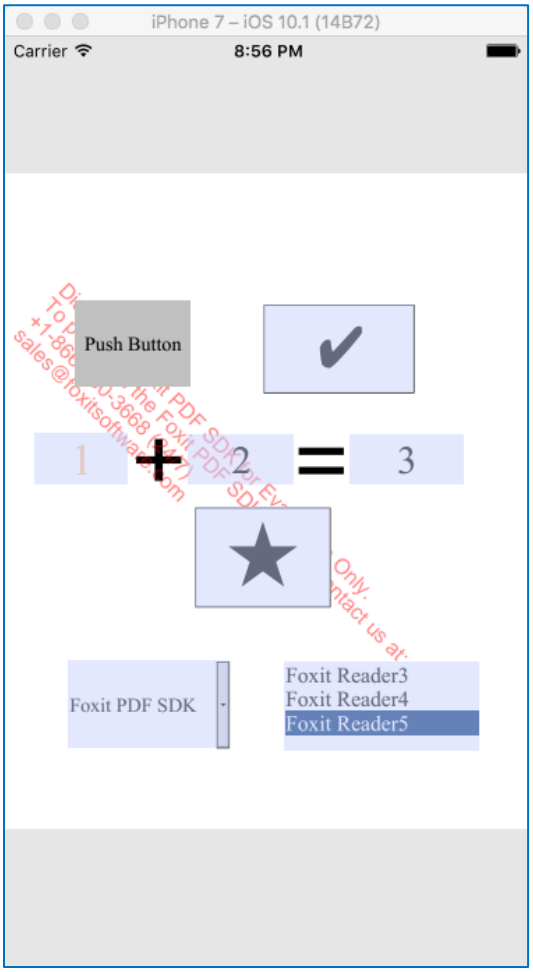


Figure 2-52

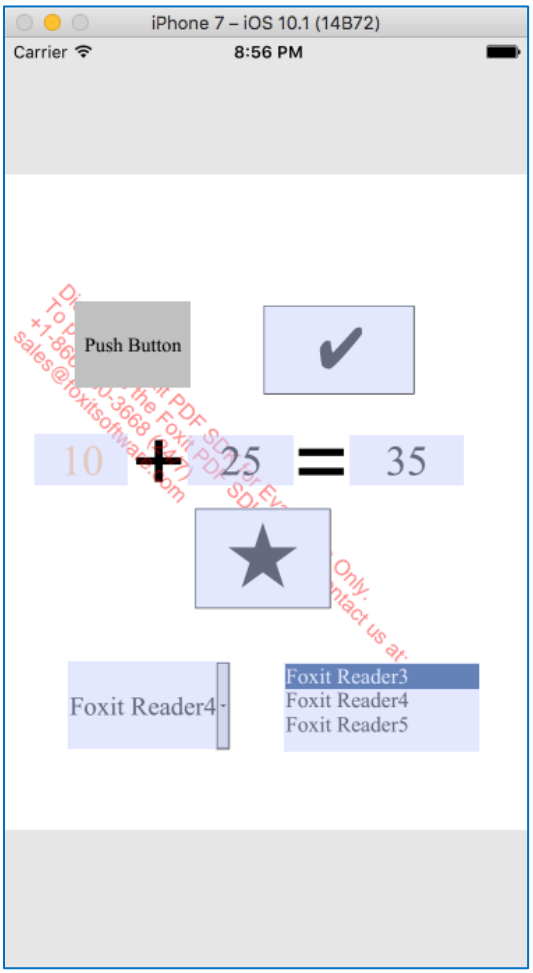


Figure 2-53

Amazing! We have realized the form filling feature in the *test_swift* project without adding any extra code related to forms. Alright, it is just a simple example which allows you to fill a PDF Form file, for further research about form, you can refer to the "complete_pdf_viewer" demo.

2.5.6 Add support for Text Search

Text search, annotations and related features are associated more closely with the user interface and as such require a slightly different approach from the form filling feature. We need to write some extra code to load the feature module and trigger the feature.

In this section, we will add support for text search and also extend the simple iOS app in the section "Displaying a PDF document". For annotations, you can refer to the "Add support for Annotations" section.

For simplicity, we will add a button on the main interface and use the button click event to quickly experience the text search feature. Just follow the steps below:

Step 1: Refer to "Preparatory work" and "Add code to instantiate a UIExtensionsManager object and set it to PDFViewCtrl" in the section "Add support for Form Filling" to add the same configuration and code in the *test_objc* project.

Step 2: In the "ViewController.swift" file, we are now going to add the code necessary for triggering the search functionality. The required code additions are shown below and further down you will find a full example of what the "ViewController.swift" file should look like.

Register the search event listener.

```
class ViewController: UIViewController, ISearchEventListener { }
...
extensionsManager.register(self)
```

Instantiate a Button object, add the click event, and set it to the root view.

```
var searchButton: UIButton!

...
searchButton = UIButton(frame: CGRect(x: Int(280), y: Int(25), width: Int(80), height:
Int(40)))
searchButton.backgroundColor = UIColor.gray
searchButton.setTitle("Search", for: .normal)
searchButton.addTarget(self, action: #selector(self.showSearchBar), for: .touchUpInside)
self.view.addSubview(searchButton)
```

The searchButton click event:

```
func showSearchBar() {
     if (extensionsManager.currentAnnot != nil) {

        extensionsManager.currentAnnot = nil
     }
        extensionsManager.showSearchBar(true)
}
```

The search event listener:

```
func onSearchStarted() {
    searchButton.isHidden = true;
}

func onSearchCanceled() {
    searchButton.isHidden = false;
}
```

The whole update of ViewController.swift is as follows:

```
import UIKit
```

```
class ViewController: UIViewController, ISearchEventListener{

    var extensionsManager: UIExtensionsManager!
    var searchButton: UIButton!

    override func viewDidLoad() {
        super.viewDidLoad()

        // Get the path of a PDF.
        let pdfPath = Bundle.main.path(forResource: "Sample", ofType: "pdf")!

        // Initialize a PDFDoc object with the path to the PDF file.
        let doc = FSPDFDoc.create(fromFilePath:pdfPath)
        if e_errSuccess != doc?.load(nil) {
            return
        }

        // Initialize a FSPDFViewCtrl object with the size of the entire screen.
        var pdfViewCtrl: FSPDFViewCtrl!
        pdfViewCtrl = FSPDFViewCtrl(frame:self.view.bounds)

        // Set the document to display.
        pdfViewCtrl.setDoc(doc)

        // Add the pdfViewCtrl to the root view.
        self.view.insertSubview(pdfViewCtrl, at: 0)

        // Initialize a UIExtensionsManager object and set it to pdfViewCtrl.
        extensionsManager = UIExtensionsManager(pdfViewControl: pdfViewCtrl)
        pdfViewCtrl.extensionsManager = extensionsManager;

        // Register the search event listener.
        extensionsManager.register(self)

        // Instantiate a Button object and add the click event.
        searchButton = UIButton(frame: CGRect(x: Int(280), y: Int(25), width: Int(80), height:
Int(40)))
        searchButton.backgroundColor = UIColor.gray
        searchButton.setTitle("Search", for: .normal)
        searchButton.addTarget(self, action: #selector(self.showSearchBar),
for: .touchUpInside)

        // Add the searchButton to the root view.
        self.view.addSubview(searchButton)

    }

    // searchButton click event
    func showSearchBar() {
        if (extensionsManager.currentAnnot != nil) {

            extensionsManager.currentAnnot = nil
        }
        extensionsManager.showSearchBar(true)
    }


    // ISearchEventListener
    func onSearchStarted() {
```

```
        searchButton.isHidden = true;
    }


    func onSearchCanceled() {
        searchButton.isHidden = false;
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

Note *For this project, we use "Sample.pdf" as the test file. You can use any PDF file, just remember to add it to the project, and change the file path in "ViewController.swift".*

Now that we have finished adding support for text search into the project, let's run it on an iPhone 7 Simulator. You will see that the "Sample.pdf" document is automatically displayed as shown in Figure 2-54.

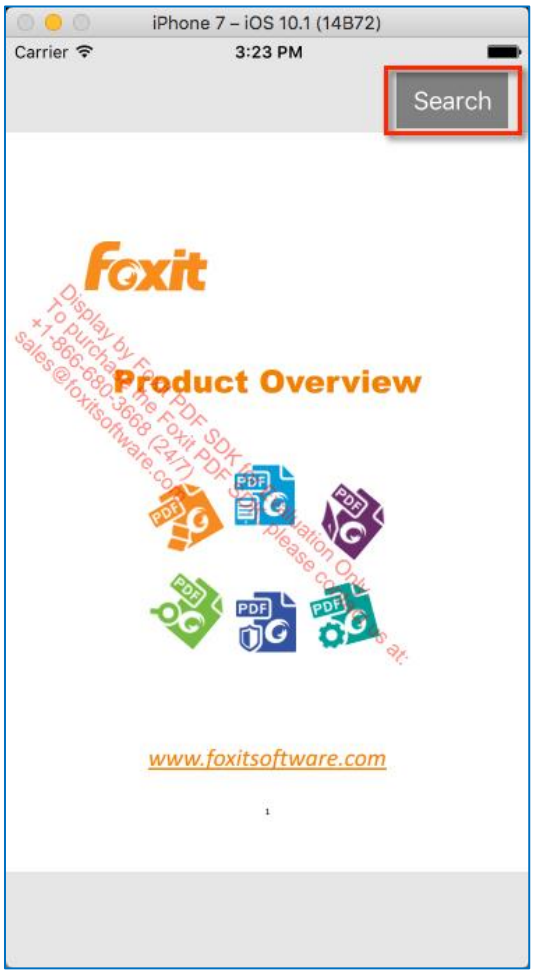


Figure 2-54

Click on the "**Search**" button, you can search anything you like. For example, input "Foxit", press "Enter", and then all of the search results will be listed as shown in Figure 2-55.

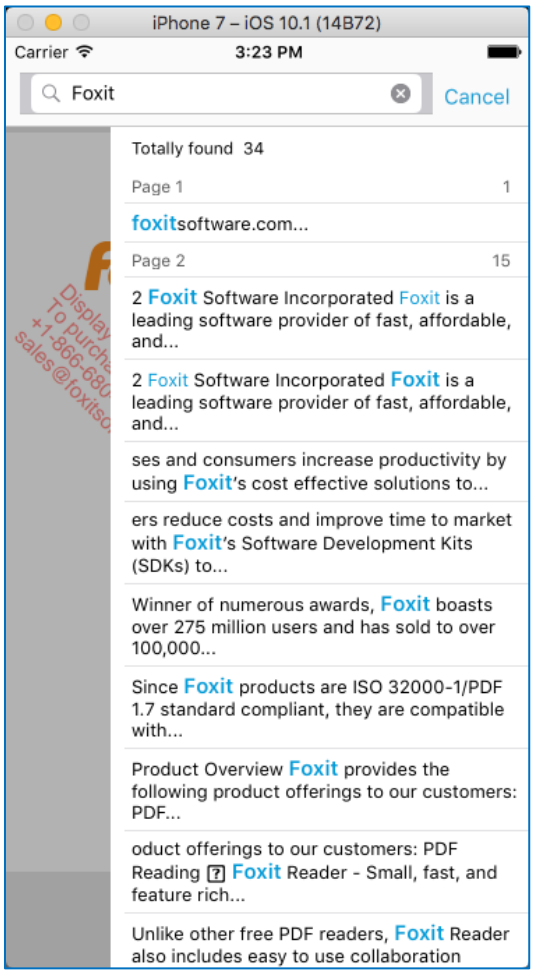


Figure 2-55

Click any result in the list to jump to the specific location. Here, we click the result in the second paragraph in the list, and then it will jump to the place where the result is, and the word will be highlighted as shown in Figure 2-56 (zoom in on the page to see it clearly). You can click the previous or next button to find the previous or next search result.

Figure 2-56

You can refer to the above code or the demos found in the download package to add support for outline, annotations, and any other available features that you wish to add.

2.5.7 Add support for Annotations

Foxit MobilePDF SDK includes a wide variety of standard annotations, and each of them uses the similar way to be added into the project. In this section, we will show you how to add support for the common used annotations which are provided in the Complete PDF Viewer demo, and will take the highlight annotation as an example to set forth the steps in detail. For other common used annotations, we only list the core code, for more details, please refer to the highlight section.

2.5.7.1 Highlight

We will add the highlight feature to the simple iOS app in the section "Displaying a PDF document", and like previous added features, we also add a button on the main interface and use the button click event to quickly experience the highlight feature. Just follow the steps below:

Step 1: Refer to "Preparatory work" and "Add code to instantiate a UIExtensionsManager object and set it to PDFViewCtrl" in the section "Add support for Form Filling" to add the same configuration and code in the *test_objc* project.

Step 2: Left-click the *test_swift* project, navigate to the **Build Settings** tab, find the **Search Paths** section**.** Add the path of the **uiextensions** folder to **Header Search Paths** in the **Build Settings** tab as shown in Figure 2-57. (In this project, we assume that you have copied the **uiextensions** folder from the "libs/uiextensions_src" of the download package to "test_swift".)

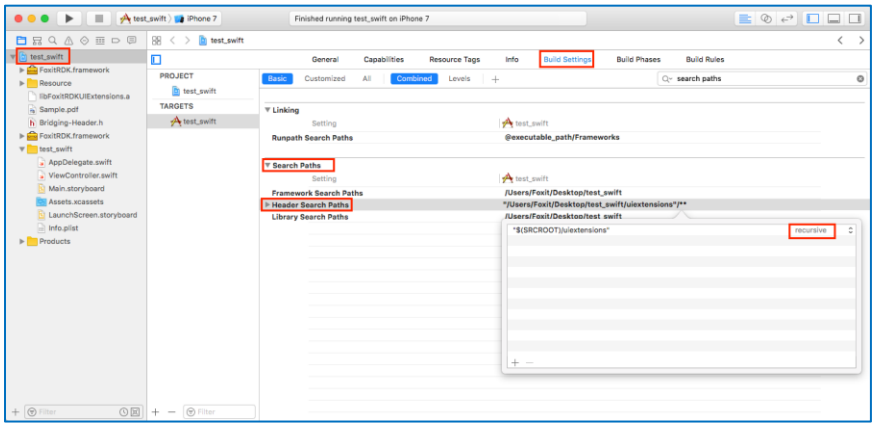


Figure 2-57

Step 3: Open the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/TextMarkup/TextMKToolHandler.h"
```

Step 4: In the "ViewController.swift" file, we are now going to add the code necessary for triggering the search functionality. The required code additions are shown below and further down you will find a full example of what the "ViewController.swift" file should look like.

Instantiate a Button object, add the click event, and set it to the root view.

```
var highlightButton: UIButton!
...
highlightButton = UIButton(frame: CGRect(x: Int(280), y: Int(25), width: Int(80), height:
Int(40)))
highlightButton.backgroundColor = UIColor.gray
```

```
highlightButton.setTitle("Highlight", for: .normal)
highlightButton.addTarget(self, action: #selector(self.highlightClick), for: .touchUpInside)
self.view.addSubview(highlightButton)
```

Handle the button click event. Get a MKToolHandler object from UIExtensionManager (the MKToolHandler is already instantiated in UIExtensionManager), set the type to highlight, and set it as the current tool handler.

```
var mkToolHandler: MKToolHandler!
...

func highlightClick() {
    mkToolHandler = extensionsManager.getToolHandler(byName:Tool_Markup) as! MKToolHandler!
    mkToolHandler.type = e_annotHighlight
    extensionsManager.currentToolHandler = mkToolHandler
}
```

The whole update of ViewController.swift is as follows:

```
import UIKit

class ViewController: UIViewController {

    var extensionsManager: UIExtensionsManager!
    var highlightButton: UIButton!
    var mkToolHandler: MKToolHandler!

    override func viewDidLoad() {
        super.viewDidLoad()

        // Get the path of a PDF.
        let pdfPath = Bundle.main.path(forResource: "Sample", ofType: "pdf")!

        // Initialize a PDFDoc object with the path to the PDF file.
        let doc = FSPDFDoc.create(fromFilePath:pdfPath)
        if e_errSuccess != doc?.load(nil) {
            return
        }

        // Initialize a FSPDFViewCtrl object with the size of the entire screen.
        var pdfViewCtrl: FSPDFViewCtrl!
        pdfViewCtrl = FSPDFViewCtrl(frame:self.view.bounds)

        // Set the document to display.
        pdfViewCtrl.setDoc(doc)
```

```
        // Add the pdfViewCtrl to the root view.
        self.view.insertSubview(pdfViewCtrl, at: 0)

        // Initialize a UIExtensionsManager object and set it to pdfViewCtrl
        extensionsManager = UIExtensionsManager(pdfViewControl: pdfViewCtrl)
        pdfViewCtrl.extensionsManager = extensionsManager;

        // Initialize a Button object and add the click event.
        highlightButton = UIButton(frame: CGRect(x: Int(280), y: Int(25), width: Int(80),
height: Int(40)))
        highlightButton.backgroundColor = UIColor.gray
        highlightButton.setTitle("Highlight", for: .normal)
        highlightButton.addTarget(self, action: #selector(self.highlightClick),
for: .touchUpInside)

        // Add the highlightButton to the root view.
        self.view.addSubview(highlightButton)

    }

    // Highlight Button click event.
    func highlightClick() {
        mkToolHandler = extensionsManager.getToolHandler(byName:Tool_Markup) as! MKToolHandler!
        mkToolHandler.type = e_annotHighlight
        extensionsManager.currentToolHandler = mkToolHandler
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

Note *For this project, we use "Sample.pdf" as the test file. You can use any PDF file, just remember to add it to the project, and change the file path in "ViewController.swift".*

Now that we have finished adding support for highlight into the project, let's run it on an iPhone 7 Simulator. You will see that the "Sample.pdf" document is automatically displayed (see Figure 2-58).

Figure 2-58

In order to see the highlight result clearly, swipe right to the next page, click the "**Highlight**" button, and drag over text to highlight the selected text (see Figure 2-59).

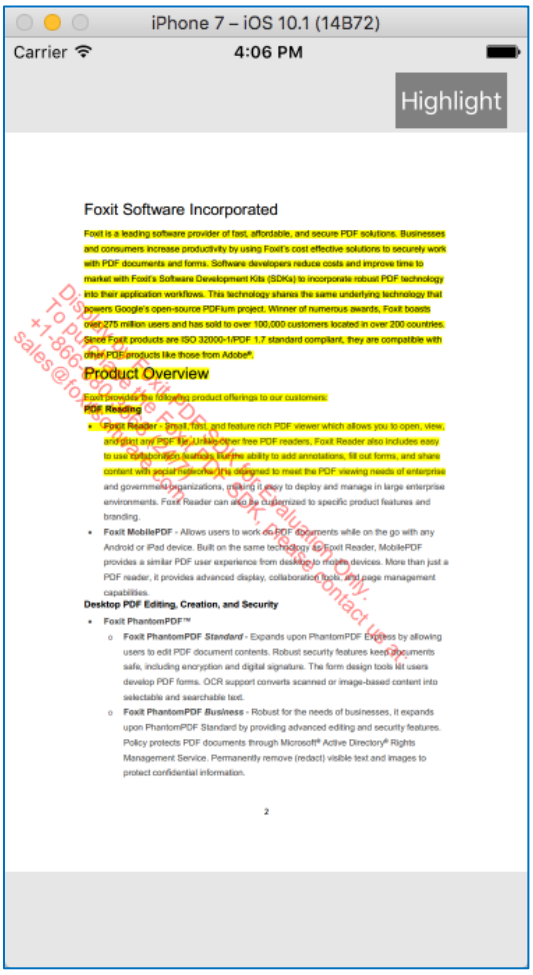


Figure 2-59

2.5.7.2 Underline

Adding support for underline is similar to add support for highlight. The core point is to get a MKToolHandler object from UIExtensionManager (the MKToolHandler is already instantiated in UIExtensionManager), set the type to underline, and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for underline. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/TextMarkup/TextMKToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var mkToolHandler: MKToolHandler!
...

func underlineClick() {
```

```
    mkToolHandler = extensionsManager.getToolHandler(byName:Tool_Markup) as! MKToolHandler!
    mkToolHandler.type = e_annotUnderline
    extensionsManager.currentToolHandler = mkToolHandler
}
```

2.5.7.3 Squiggly

Adding support for squiggly is similar to add support for highlight. The core point is to get a MKToolHandler object from UIExtensionManager (the MKToolHandler is already instantiated in UIExtensionManager), set the type to squiggly, and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for squiggly. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/TextMarkup/TextMKToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var mkToolHandler: MKToolHandler!
...

func squigglyClick() {
    mkToolHandler = extensionsManager.getToolHandler(byName:Tool_Markup) as! MKToolHandler!
    mkToolHandler.type = e_annotSquiggly
    extensionsManager.currentToolHandler = mkToolHandler
}
```

2.5.7.4 Strikeout

Adding support for strikeout is similar to add support for highlight. The core point is to get a MKToolHandler object from UIExtensionManager (the MKToolHandler is already instantiated in UIExtensionManager), set the type to strikeout, and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for strikeout. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/TextMarkup/TextMKToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var mkToolHandler: MKToolHandler!
...

func strikeoutClick() {
    mkToolHandler = extensionsManager.getToolHandler(byName:Tool_Markup) as! MKToolHandler!
    mkToolHandler.type = e_ annotStrikeOut
    extensionsManager.currentToolHandler = mkToolHandler
```

```
}
```

2.5.7.5 Insert text

Adding support for insert text is similar to add support for highlight. The core point is to get an InsertToolHandler object from UIExtensionManager (the InsertToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for insert text. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/Caret/InsertToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var insertToolHandler: InsertToolHandler!
...

func insertClick() {
    insertToolHandler = extensionsManager.getToolHandler(byName:Tool_Insert) as!
InsertToolHandler!
    extensionsManager.currentToolHandler = insertToolHandler
}
```

2.5.7.6 Replace text

Adding support for replace text is similar to add support for highlight. The core point is to get a ReplaceToolHandler object from UIExtensionManager (the ReplaceToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for replace text. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/Caret/ReplaceToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var replaceToolHandler: ReplaceToolHandler!
...

func replaceClick() {
    replaceToolHandler = extensionsManager.getToolHandler(byName:Tool_Replace) as!
ReplaceToolHandler!
    extensionsManager.currentToolHandler = replaceToolHandler
}
```

2.5.7.7 Line

Line and arrow tools are implemented in LineToolHandler. So, the core point to add support for line is to get a LineToolHandler object from UIExtensionManager (the LineToolHandler is already instantiated in UIExtensionManager), set the type to line (default type is also line), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for line. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/Line/LineToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var lineToolHandler: LineToolHandler!
...

func lineClick() {
    lineToolHandler = extensionsManager.getToolHandler(byName:Tool_Line) as! LineToolHandler!
    lineToolHandler.type = e_annotLine; // default type is also line.
    extensionsManager.currentToolHandler = lineToolHandler
}
```

2.5.7.8 Arrow

Line and arrow tools are implemented in LineToolHandler. So, the core point to add support for arrow is to get a LineToolHandler object from UIExtensionManager (the LineToolHandler is already instantiated in UIExtensionManager), set the "isArrowLine" to "YES", and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for arrow. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/Line/LineToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var lineToolHandler: LineToolHandler!
...

func arrowClick() {
    lineToolHandler = extensionsManager.getToolHandler(byName:Tool_Line) as! LineToolHandler!
    lineToolHandler.isArrowLine = true
    extensionsManager.currentToolHandler = lineToolHandler
}
```

2.5.7.9 Square

Square and circle tools are implemented in ShapeToolHandler. So, the core point to add support for square is to get a ShapeToolHandler object from UIExtensionManager (the ShapeToolHandler is already instantiated in UIExtensionManager), set the type to square, and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for square. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/Line/ShapeToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var shapeToolHandler: ShapeToolHandler!
...

func squareClick() {
    shapeToolHandler = extensionsManager.getToolHandler(byName:Tool_Shape) as!
ShapeToolHandler!
    shapeToolHandler.type = e_annotSquare
    extensionsManager.currentToolHandler = shapeToolHandler
}
```

2.5.7.10 Circle

Square and circle tools are implemented in ShapeToolHandler. So, the core point to add support for circle is to get a ShapeToolHandler object from UIExtensionManager (the ShapeToolHandler is already instantiated in UIExtensionManager), set the type to circle, and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for circle. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/Line/ShapeToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var shapeToolHandler: ShapeToolHandler!
...

func circleClick() {
    shapeToolHandler = extensionsManager.getToolHandler(byName:Tool_Shape) as!
ShapeToolHandler!
    shapeToolHandler.type = e_annotCircle
    extensionsManager.currentToolHandler = shapeToolHandler
}
```

2.5.7.11 Pencil

Adding support for pencil is similar to add support for highlight. The core point is to get a PencilToolHandler object from UIExtensionManager (the PencilToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for pencil. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/Pencil/Pencil/PencilToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var pencilToolHandler: PencilToolHandler!
...

func pencilClick() {
    pencilToolHandler = extensionsManager.getToolHandler(byName:Tool_Pencil) as!
PencilToolHandler!
    extensionsManager.currentToolHandler = pencilToolHandler
}
```

2.5.7.12 Eraser

Eraser tool is generally used in combination with the pencil tool. The core point to add support for eraser is to get an EraseToolHandler object from UIExtensionManager (the EraseToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for eraser. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/Pencil/Erase/EraseToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var eraseToolHandler: EraseToolHandler!
...

func eraserClick() {
    eraseToolHandler = extensionsManager.getToolHandler(byName:Tool_Eraser) as!
EraseToolHandler!
    extensionsManager.currentToolHandler = eraseToolHandler
}
```

2.5.7.13 Typewriter

Adding support for typewriter is similar to add support for highlight. The core point is to get a FtToolHandler object from UIExtensionManager (the FtToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for typewriter. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/Freetext/FtToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var ftToolHandler: FtToolHandler!
...

func typewriterClick() {
    ftToolHandler = extensionsManager.getToolHandler(byName:Tool_Freetext) as! FtToolHandler!
    extensionsManager.currentToolHandler = ftToolHandler
}
```

2.5.7.14 Note

Adding support for note is similar to add support for highlight. The core point is to get a NoteToolHandler object from UIExtensionManager (the NoteToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for note. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/Note/NoteToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var noteToolHandler: NoteToolHandler!
...

func noteClick() {
    noteToolHandler = extensionsManager.getToolHandler(byName:Tool_Note) as! NoteToolHandler!
    extensionsManager.currentToolHandler = noteToolHandler
}
```

2.5.7.15 Stamp

Adding support for stamp is similar to add support for highlight. The core point is to get a StampToolHandler object from UIExtensionManager (the StampToolHandler is already instantiated in

UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for stamp. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/Stamp/StampToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var stampToolHandler: StampToolHandler!
...

func stampClick() {
    stampToolHandler = extensionsManager.getToolHandler(byName:Tool_ Stamp) as!
StampToolHandler!
    extensionsManager.currentToolHandler = stampToolHandler
}
```

2.5.7.16 Attachment

Adding support for attachment is similar to add support for highlight. The core point is to get an AttachmentToolHandler object from UIExtensionManager (the AttachmentToolHandler is already instantiated in UIExtensionManager), and set it as the current tool handler. You can refer to "Highlight" to see the detailed process to add support for attachment. Following is the key steps:

In the "Bridging-Header.h" file, import the following header file:

```
#import "uiextensions/Attachment/AttachmentToolHandler.h"
```

In the "ViewController.swift" file, the core code is

```
var attachmentToolHandler: AttachmentToolHandler!
...

func attachmentClick() {
    attachmentToolHandler = extensionsManager.getToolHandler(byName:Tool_Attachment) as!
AttachmentToolHandler!
    extensionsManager.currentToolHandler = attachmentToolHandler
}
```

3 Customizing the UI Implementation

Customizing the UI implementation is straightforward. Foxit MobilePDF SDK provides the source code of the UI Extensions Component that contains ready-to-use UI module implementations, which lets the developers have full control of styling the appearance as desired.

There is one thing to take note of. The source code of the UI Extensions Component is written in Objective-C, so you need to use Objective-C to modify the UI layout. If you are a Swift developer and not already familiar with Objective-C, you might only be able to customize the UI appearance that does not need writing code, such as icons and other UI resources.

To customize the UI implementation, you need to follow these steps:

First, add the following required files into your app.

- ***FoxitRDK.framework*** – The framework that includes the Foxit MobilePDF SDK dynamic library and associated header files. It can be found in the "libs" folder.
- ***uiextensions*** *project* – It is an open source library that contains some ready-to-use UI module implementations, which can help developers rapidly embed a fully functional PDF reader into their iOS app. Of course, developers are not forced to use the default UI, they can freely customize and design the UI for their specific apps through the "uiextensions" project. It can be found in the "libs/uiextensions_src" folder.

  Note *The built-in UI customization can be done in the* ***uiextensions*** *project, and then you can add the new* ***libFoxitRDKUIExtensions.a*** *library generated by the modified* ***uiextensions*** *project to your app instead of the whole* ***uiextensions*** *project.*

Second, find the specific code or images related to the UI that you want to customize in the ***uiextensions*** project, then modify them based on your requirements.

Now, for your convenience, we will show you how to customize the UI implementation in "**viewer_ctrl_demo**" project found in the "samples" folder.

UI Customization Example

Step 1: Add the ***uiextensions*** project into the demo**.**

Note *We will add the* ***uiextensions*** *project to the demo which is convenient for us to see the custom results. The demo already includes* ***FoxitRDK.framework****, so we just need to add the* ***uiextensions*** *project.*

Load the "**viewer_ctrl_demo**" project in Xcode. Drag-and-drop "**uiextensions.xcodeproj**" found in the "libs/uiextensions_src" of the download package into the "**viewer_ctrl_demo**" project as shown in Figure 3-1.

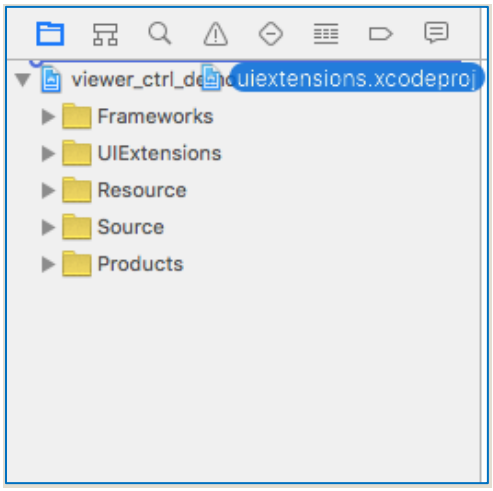


Figure 3-1

Then, it will pop up a dialog box which prompts you whether to save the project in a new workspace as shown in Figure 3-2. Click **Save**.

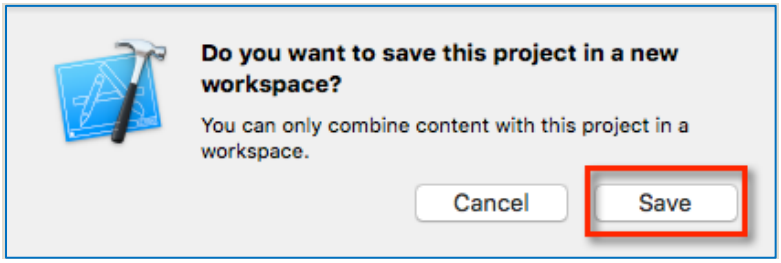


Figure 3-2

Save the workspace to the "samples" folder, and name "custom_viewer" as shown in Figure 3-3. Click **Save**.

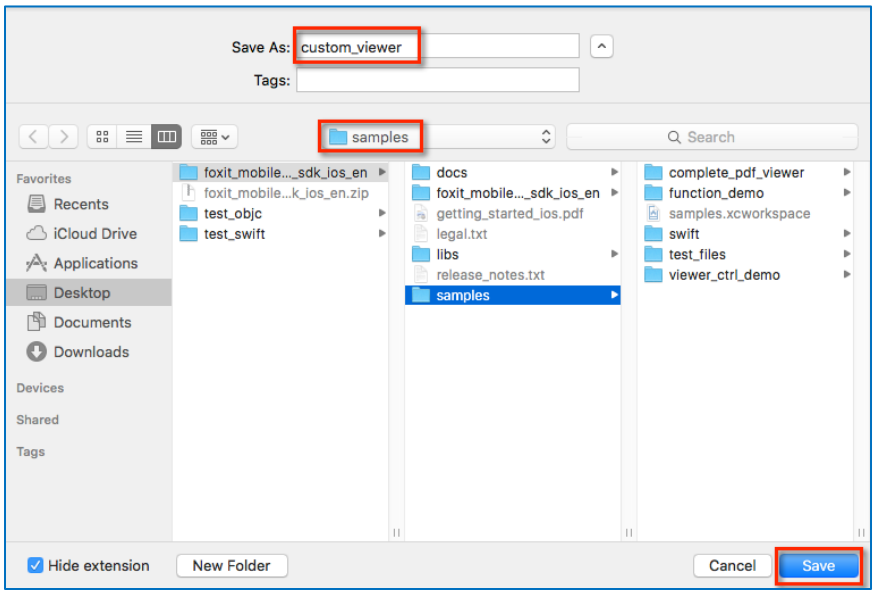


Figure 3-3

Now, the workspace looks like the Figure 3-4.

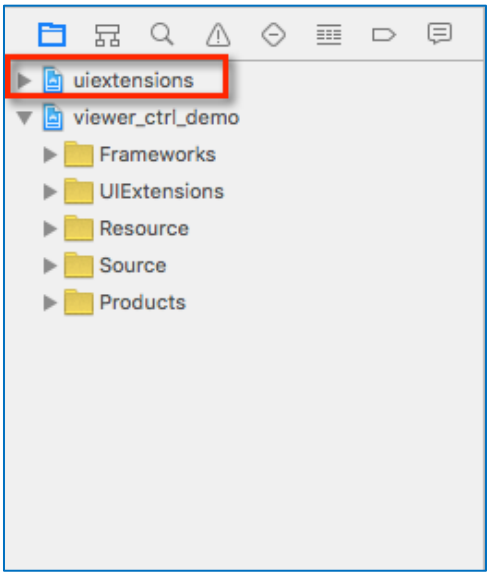


Figure 3-4

Congratulations! You have completed the first step.

Step 2: Find and modify the code or images related to the UI that you want to customize.

Now we will show you a simple example that changes one button's icon in the search panel as shown in Figure 3-5.

Figure 3-5

To replace the icon we only need to find the place which stores the icon for this button, then use another icon with the same name to replace it.

Note *Foxit MobilePDF SDK provides three sets of icons for different devices to make sure that your apps can run smoothly on every device. There are three folders used to store the image resources as follows:*

- ✓ *Image: used for older devices with non-Retina display (e.g. iPad 2).*
- ✓ *Image2x: used for iPhone 4/4s/5/5s/6/6s/7.*
- ✓ *Image3x: used for iPhone 6/6s/7 Plus, which has the highest resolution.*

An iPhone 7 Simulator will be used as an example to run the demo. In the project, click "**Resource** -> **png** -> **image2x** -> **Search**" as shown in Figure 3-6. It's easy to find the image that we want to replace. The resource files are stored according to the features, so you can locate the related code through the icon's name.

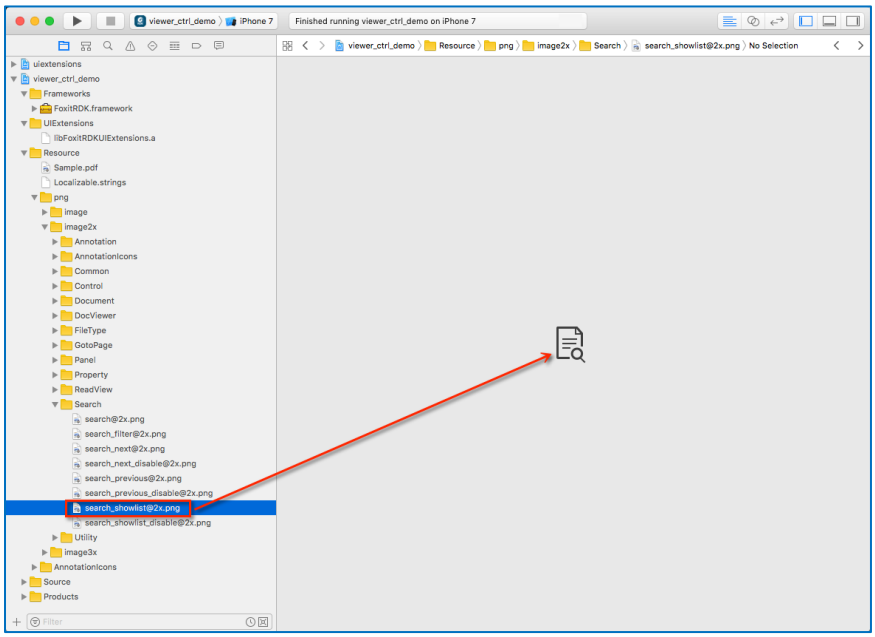


Figure 3-6

Right now, just replace "search_showlist@2x.png" with your own icon in the "libs/uiextensions_src/uiextensions/Resource/png/image2x" folder. For example, we use the icon of the top search button (search@2x.png) to replace it.

After replacing, firstly build and run the ***uiextensions_aggregate*** *project* as shown in Figure 3-7.

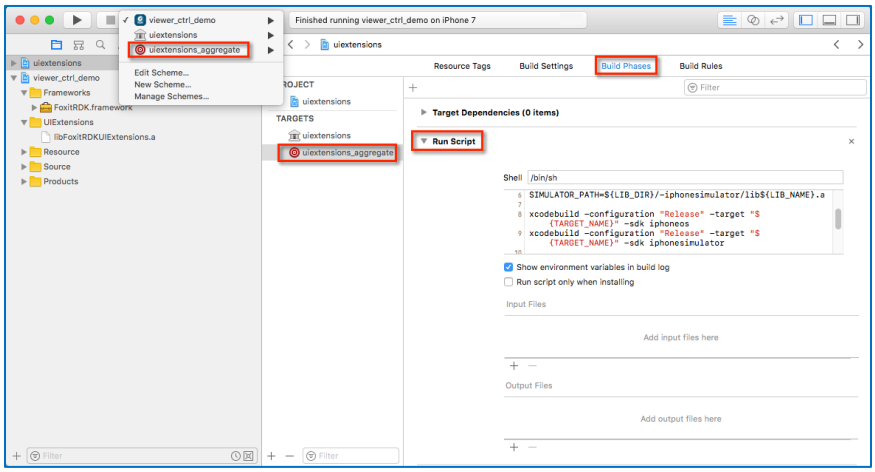


Figure 3-7

Here, we build and run ***uiextensions_aggregate****, then* the generated library will include the libraries for both simulator and iOS device. If you choose ***uiextensions*** *and run it on a simulator, then the generated library is a simulator library which can only be used for simulator.*

Note *The "**libFoxitRDKUIExtensions.a**" library in the "libs" folder of the download package is a universal static library which includes the libraries for both simulator and iOS device. It will be overwritten after building the "uiextensions" project successfully. In the "uiextension" project, it has already added the scripts to generate the universal static library as shown in Figure 3-7.*

Then build and run the "**viewer_ctrl_demo**" project. After building successfully, try the search feature and we can see that the icon of the bottom search button has changed as shown in Figure 3-8.

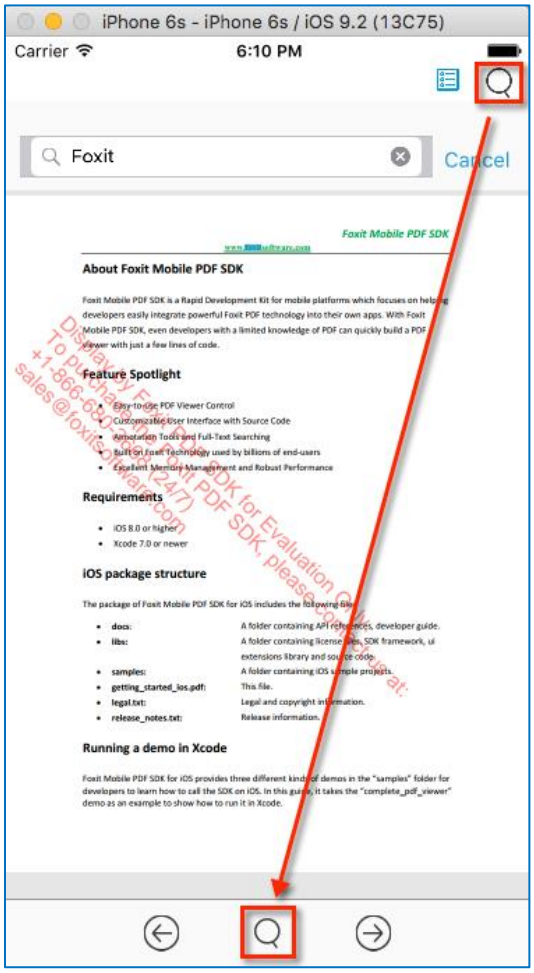


Figure 3-8

This is just a simple example to show how to customize the UI implementation. You can refer to it and feel free to customize and design the UI for your specific apps through the ***uiextensions*** project.

4 Creating a Custom Tool

With Foxit MobilePDF SDK, creating a custom tool is a simple process. There are several tools implemented in the UI Extensions Component already. These tools can be used as a base for developers to build upon or use as a reference to create a new tool. In order to create your own tool quickly, we suggest you take a look at the ***uiextension*** project found in the "libs" folder.

To create a new tool, the most important step is to declare a class that implements the "**IToolHandler**" interface.

In this section, we will make a Regional Screenshot Tool to show how to create a custom tool with Foxit MobilePDF SDK. This tool can help the users who only want to select an area in a PDF page to capture, and then save it as an image. Now, let's do it.

4.1 Create a Regional Screenshot Tool in Objective-C

For convenience, we will build this tool based on the "**viewer_ctrl_demo**" project found in the "samples" folder. Steps required for implementing this tool are as follows:

- Create a class named **ScreenCaptureToolHandler** that implements the "**IToolHandler**" interface.
- Handle **onPageViewLongPress** and **onDraw** events.
- Instantiate a **ScreenCaptureToolHandler** object, and then register it to the extensions manager.
- Set the **ScreenCaptureToolHandler** object as the current tool handler.

Step 1: Create a class named **ScreenCaptureToolHandler** that implements the "**IToolHandler**" interface.

a) Load the "**viewer_ctrl_demo**" project in Xcode. Create a class named "**ScreenCaptureToolHandler**" in the "Source" folder, and create the corresponding header file.

b) Let the **ScreenCaptureToolHandler** class implement the I**ToolHandler** interface as follows:

```
@interface ScreenCaptureToolHandler : NSObject<IToolHandler>
```

Step 2: Handle **onTouchEvent** and **onDraw** events.

Update ScreenCaptureToolHandler.h as follows:

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

```
#import <FoxitRDK/FSPDFViewControl.h>
#import "../../../libs/uiextensions_src/uiextensions/UIExtensionsManager.h"
@protocol IToolHandler;
@class TaskServer;

@interface ScreenCaptureToolHandler : NSObject<IToolHandler>

- (instancetype)initWithUIExtensionsManager:(UIExtensionsManager*)extensionsManager
taskServer:(TaskServer*)taskServer;
@end
```

Update ScreenCaptureToolHandler.m as follows:

```
#import "ScreenCaptureToolHandler.h"
#import "../../../libs/uiextensions_src/uiextensions/UIExtensionsManager.h"
#import <ImageIO/ImageIO.h>
#import <ImageIO/CGImageDestination.h>
#import <MobileCoreServices/UTCoreTypes.h>

@interface ScreenCaptureToolHandler ()

@end

@implementation ScreenCaptureToolHandler {
    UIExtensionsManager* _extensionsManager;
    FSPDFViewCtrl* _pdfViewCtrl;
    TaskServer* _taskServer;

    CGPoint startPoint;
    CGPoint endPoint;

}
@synthesize type;

- (instancetype)initWithUIExtensionsManager:(UIExtensionsManager*)extensionsManager
taskServer:(TaskServer*)taskServer
{
    self = [super init];
    if (self) {
        _extensionsManager = extensionsManager;
        _pdfViewCtrl = extensionsManager.pdfViewCtrl;
        _taskServer = taskServer;
    }
    return self;
}

-(NSString*)getName
{
    return @" ";
}

-(BOOL)isEnabled
{
    return YES;
}

-(void)onActivate
{
```

```
}

-(void)onDeactivate
{

}

// Save the image to a specified path.
- (void)saveJPGImage:(CGImageRef)imageRef path:(NSString *)path
{
    NSURL *fileURL = [NSURL fileURLWithPath:path];
    CGImageDestinationRef dr = CGImageDestinationCreateWithURL((__bridge CFURLRef)fileURL,
kUTTypeJPEG , 1, NULL);

    CGImageDestinationAddImage(dr, imageRef, NULL);
    CGImageDestinationFinalize(dr);

    CFRelease(dr);
}

// Handle the PageView Gesture and Touch event
- (BOOL)onPageViewLongPress:(int)pageIndex recognizer:(UILongPressGestureRecognizer
*)recognizer
{
    if (recognizer.state == UIGestureRecognizerStateBegan)
    {
        startPoint = [recognizer locationInView:[_pdfViewCtrl getPageView:pageIndex]];
        endPoint = startPoint;
    }
    else if (recognizer.state == UIGestureRecognizerStateChanged)
    {

        endPoint = [recognizer locationInView:[_pdfViewCtrl getPageView:pageIndex]];

        // Refresh the page view, then the onDraw event will be triggered.
        [_pdfViewCtrl refresh:pageIndex];
    }
    else if (recognizer.state == UIGestureRecognizerStateEnded || recognizer.state ==
UIGestureRecognizerStateCancelled)
    {
        // Get the size of the Rect.
        CGSize size = {fabs(endPoint.x-startPoint.x), fabs(endPoint.y-startPoint.y)};
        CGPoint origin = {startPoint.x<endPoint.x?startPoint.x:endPoint.x,
startPoint.y<endPoint.y?startPoint.y:endPoint.y};
        // Get the Rect.
        CGRect rect = {origin, size};

        int newDibWidth = rect.size.width;
        int newDibHeight = rect.size.height;
        if (newDibWidth < 1 || newDibHeight < 1)
        {
            return YES;
        }

        UIView* pageView = [_pdfViewCtrl getPageView:pageIndex];
        CGRect bound = pageView.bounds;

        // Create a bitmap with the size of the selected area.
        int imgSize = newDibWidth*newDibHeight*4;
```

```
        void* pBuff = malloc(newDibWidth*newDibHeight*4);
        FSBitmap* fsbitmap = [FSBitmap create:newDibWidth height:newDibHeight format:e_dibArgb
buffer:pBuff pitch:newDibWidth*4];
        [fsbitmap fillRect:0xFFFFFFFF rect:nil];
        FSRenderer* fsrenderer = [FSRenderer create:fsbitmap rgbOrder:YES];
        FSPDFPage* page = [_pdfViewCtrl.currentDoc getPage:pageIndex];

        // Calculate the display matrix.
        FSMatrix* fsmatrix = [page getDisplayMatrix:-rect.origin.x yPos:-rect.origin.y
xSize:bound.size.width ySize:bound.size.height rotate:0];

        // Set the render content, then start to render the selected area to the bitmap.
        [fsrenderer setRenderContent:e_renderPage|e_renderAnnot];
        [fsrenderer startRender:page matrix:fsmatrix pause:nil];
        [fsrenderer continueRender];

        // Convert FSBitmap to CGImage.
        CGDataProviderRef provider = CGDataProviderCreateWithData(NULL, pBuff, imgSize, nil);
        CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
        CGBitmapInfo bitmapInfo = kCGBitmapByteOrderDefault|kCGImageAlphaLast;

        CGImageRef image = CGImageCreate(newDibWidth,newDibHeight, 8, 32, newDibWidth * 4,
                                         colorSpace, bitmapInfo,
                                         provider, NULL, YES, kCGRenderingIntentDefault);

        // Save the image to a specified path.
        NSString* jpgPath =@"/Users/Foxit/Desktop/ScreenCapture.jpg";
        [self saveJPGImage:image path:jpgPath];

        UIAlertView *alert = [[UIAlertView alloc]initWithTitle:@""
                                                       message:@" The selected area was saved
as a JPG stored in the /Users/Foxit/Desktop/ScreenCapture.jpg" delegate:nil
cancelButtonTitle:NSLocalizedString(@"OK", @"OK") otherButtonTitles:nil];
        [alert show];

        return YES;
    }
    return YES;
}

// Handle the drawing event.
-(void)onDraw:(int)pageIndex inContext:(CGContextRef)context
{
    if ([_extensionsManager getCurrentToolHandler] != self) {
        return;
    }

    CGContextSetLineWidth(context, 2);
    CGContextSetLineCap(context, kCGLineCapSquare);
    UIColor *color = [UIColor redColor];
    CGContextSetStrokeColorWithColor(context, [color CGColor]);
    CGPoint points[] = {startPoint,CGPointMake(endPoint.x,
startPoint.y),endPoint,CGPointMake(startPoint.x, endPoint.y)};
    CGContextAddLines(context,points,4);
    CGContextClosePath(context);
    CGContextStrokePath(context);
}

- (BOOL)onPageViewTap:(int)pageIndex recognizer:(UITapGestureRecognizer *)recognizer
```

```
{
    return NO;
}

- (BOOL)onPageViewPan:(int)pageIndex recognizer:(UIPanGestureRecognizer *)recognizer
{
    return NO;

}

- (BOOL)onPageViewShouldBegin:(int)pageIndex recognizer:(UIGestureRecognizer
*)gestureRecognizer
{
    if (self != [_extensionsManager getCurrentToolHandler]) {
        return NO;
    }
    return YES;
}


- (BOOL)onPageViewTouchesBegan:(int)pageIndex touches:(NSSet*)touches
withEvent:(UIEvent*)event
{
    return NO;
}

- (BOOL)onPageViewTouchesMoved:(int)pageIndex touches:(NSSet *)touches withEvent:(UIEvent
*)event
{
    return NO;
}

- (BOOL)onPageViewTouchesEnded:(int)pageIndex touches:(NSSet *)touches withEvent:(UIEvent
*)event
{
    return NO;
}

- (BOOL)onPageViewTouchesCancelled:(int)pageIndex touches:(NSSet *)touches withEvent:(UIEvent
*)event
{
    return NO;
}

@end
```

Note *In the above code, you should specify an existing path to save the image. Here, the path is "@"/Users/Foxit/Desktop/ScreenCapture.jpg"", please replace it with a valid path.*

Step 3: Instantiate a **ScreenCaptureToolHandler** object and then register it to the UIExtensionsManager.

```
#import "ScreenCaptureToolHandler.h"
...

@property (nonatomic, strong) ScreenCaptureToolHandler* screencaptureToolHandler;
...
```

```
self.screencaptureToolHandler = [[[ScreenCaptureToolHandler alloc] initWithUIExtensionsManager:
self.extensionsManager taskServer:nil] autorelease];
[self.extensionsManager registerToolHandler:self.screencaptureToolHandler];
```

Step 4: Set the **ScreenCaptureToolHandler** object as the current tool handler.

```
[self.extensionsManager setCurrentToolHandler:self.screencaptureToolHandler];
```

Now, we have really finished creating a custom tool in Objective-C. In order to see what the tool looks like, we need to make it run. Just add the code referred in Step 3 and Step 4 to ViewController.m.

After finishing all of the above work, build and run the demo. An iPhone 7 Simulator will be used as an example to run the project. After building the demo successfully, long press and select a rectangular area, and then a message box will be popped up as shown in Figure 4-1. It shows where the image (selected area) was saved to.

Figure 4-1

In order to verify whether the tool captures the selected area successfully, we need to find the screenshot. Go to "desktop", we can see the image as shown in Figure 4-2.

Figure 4-2

As you can see we have successfully created a Regional Screenshot Tool in Objective-C. This is just an example to show how to create a custom tool with Foxit MobilePDF SDK. You can refer to it or the demos to develop the tools you want.

4.2 Create a Regional Screenshot Tool in Swift

For convenience, we will build this tool based on the "**viewer_ctrl_demo_swift**" project found in the "samples\swift" folder. Steps required for implementing this tool are as follows:

- Create a class named **ScreenCaptureToolHandler** that implements the "**IToolHandler**" interface.
- Handle **onPageViewLongPress** and **onDraw** events.
- Instantiate a **ScreenCaptureToolHandler** object, and then register it to the extensions manager.
- Set the **ScreenCaptureToolHandler** object as the current tool handler.

Step 1: Create a class named **ScreenCaptureToolHandler** that implements the "**IToolHandler**" interface.

a) Load the "**viewer_ctrl_demo_swift**" project in Xcode. Create a class named "**ScreenCaptureToolHandler**" in the "Source" folder.

b) Let the **ScreenCaptureToolHandler** class implement the I**ToolHandler** interface as follows:

```
class ScreenCaptureToolHandler: NSObject, IToolHandler { }
```

Step 2: Handle **onTouchEvent** and **onDraw** events.

Update **ScreenCaptureToolHandler.swift** as follows:

```
import Foundation
import MobileCoreServices
import ImageIO

class ScreenCaptureToolHandler: NSObject, IToolHandler {
    public var type: FS_ANNOTTYPE

    var extensionManager: UIExtensionsManager!
    var pdfViewCtrl: FSPDFViewCtrl!

    var startPoint = CGPoint()
    var endPoint = CGPoint()

    init(extensionsManager: UIExtensionsManager) {
        self.extensionManager = extensionsManager
        self.pdfViewCtrl = extensionsManager.pdfViewCtrl
        self.type = e_annotUnknownType
        super.init()
    }

    func getName() -> String {
        return " "
    }

    func isEnabled() -> Bool {
        return true
    }

    func onActivate() {
    }

    func onDeactivate() {
    }

    // Save the image to a specified path.
    func saveJPGImage(imageRef: CGImage, path: String) {
        let fileURL: CFURL = NSURL.fileURL(withPath: path) as CFURL
        let dr = CGImageDestinationCreateWithURL(fileURL, kUTTypeJPEG, 1, nil)!
        CGImageDestinationAddImage(dr, imageRef, nil)
        CGImageDestinationFinalize(dr)
    }

    // Handle the PageView Gesture and Touch event
    func onPageViewLongPress(_ pageIndex: Int32, recognizer: UILongPressGestureRecognizer!) ->
Bool {

        if recognizer.state == UIGestureRecognizerState.began {
            startPoint = recognizer.location(in: pdfViewCtrl.getPageView(pageIndex))
            endPoint = startPoint
        }
        else if recognizer.state == UIGestureRecognizerState.changed {

            endPoint = recognizer.location(in: pdfViewCtrl.getPageView(pageIndex))
```

```
            // Refresh the page view, then the onDraw event will be triggered.
            pdfViewCtrl.refresh(pageIndex)
        }
        else if recognizer.state == UIGestureRecognizerState.ended || recognizer.state ==
UIGestureRecognizerState.cancelled {

            // Get the size of the Rect.
            let size = CGSize(width: fabs(endPoint.x - startPoint.x), height: fabs(endPoint.y
- startPoint.y))
            let origin = CGPoint(x: (startPoint.x < endPoint.x) ? startPoint.x : endPoint.x, y:
(startPoint.y<endPoint.y) ? startPoint.y : endPoint.y)
            // Get the Rect.
            let rect = CGRect(origin: origin, size: size)

            let newDibwidth = rect.size.width
            let newDibHeight = rect.size.height
            if newDibwidth < 1 || newDibHeight < 1 {
                return true
            }

            let pageView = pdfViewCtrl.getPageView(pageIndex)
            let bound = pageView?.bounds

            // Create a bitmap with the size of the selected area.
            let imgSize = newDibwidth * newDibHeight * 4
            let capacity: Int = Int(newDibwidth) * Int(newDibHeight) * 4
            let pBuff = UnsafeMutablePointer<UInt8>.allocate(capacity: capacity)
            let pitch: Int = Int(newDibwidth) * 4
            let fsbitmap = FSBitmap.create(Int32(newDibwidth), height: Int32(newDibHeight),
format: e_dibArgb, buffer: pBuff, pitch: Int32(pitch))
            fsbitmap?.fillRect(0xFFFFFFFF, rect: nil)
            let fsrenderer = FSRenderer.create(fsbitmap, rgbOrder: true)
            let page = pdfViewCtrl.currentDoc.getPage(pageIndex)

            // Calculate the display matrix.
            let fsmatrix = page?.getDisplayMatrix(-(Int32)(rect.origin.x), yPos: -
(Int32)(rect.origin.y), xSize: Int32((bound?.size.width)!), ySize:
Int32((bound?.size.height)!), rotate: FS_ROTATION(rawValue: e_rotation0.rawValue))

            // Set the render content, then start to render the selected area to the bitmap.
            fsrenderer?.setRenderContent(e_renderPage.rawValue | e_renderAnnot.rawValue)
            fsrenderer?.startRender(page, matrix: fsmatrix, pause: nil)
            fsrenderer?.continueRender()

            // Convert FSBitmap to CGImage.
            let releaseData: CGDataProviderReleaseDataCallback = {
                (info: UnsafeMutableRawPointer?, data:UnsafeRawPointer, size:Int) -> Void in
            }

            let provider: CGDataProvider = CGDataProvider(dataInfo: nil, data: pBuff, size:
Int(imgSize), releaseData: releaseData)!
            let colorSpace = CGColorSpaceCreateDeviceRGB()
            let bitmapInfo: CGBitmapInfo = .byteOrderMask

            let image = CGImage(width: Int(newDibwidth), height: Int(newDibHeight),
bitsPerComponent: 8, bitsPerPixel: 32, bytesPerRow: Int(newDibwidth) * 4, space: colorSpace,
bitmapInfo: bitmapInfo, provider: provider, decode: nil, shouldInterpolate: true, intent:
CGColorRenderingIntent.defaultIntent)
```

```
            // Save the image to a specified path.
            let jpgPath = "/Users/Foxit/Desktop/ScreenCapture.jpg"
            self.saveJPGImage(imageRef: image!, path: jpgPath)

            let alert = UIAlertView(title: "", message: " The selected area was saved as a JPG
stored in the /Users/Foxit/Desktop/ScreenCapture.jpg", delegate: nil, cancelButtonTitle:
NSLocalizedString("OK", comment: "OK"))
            alert.show()
            return true
        }
        return true
    }

    // Handle the drawing event.
    func onDraw(_ pageIndex: Int32, in context: CGContext!) {
        context.setLineWidth(CGFloat(2))
        context.setLineCap(.square)
        let color = UIColor.red
        context.setStrokeColor(color.cgColor)
        let points = [startPoint, CGPoint(x: CGFloat(endPoint.x), y: CGFloat(startPoint.y)),
endPoint, CGPoint(x: CGFloat(startPoint.x), y: CGFloat(endPoint.y))]
        context.addLines(between: points)
        context.closePath()
        context.strokePath()
    }

    func onPageViewTap(_ pageIndex: Int32, recognizer: UITapGestureRecognizer!) -> Bool {
        return false
    }

    func onPageViewPan(_ pageIndex: Int32, recognizer: UIPanGestureRecognizer!) -> Bool {
        return false
    }

    func onPageViewShouldBegin(_ pageIndex: Int32, recognizer gestureRecognizer:
UIGestureRecognizer!) -> Bool {
        return true
    }

    func onPageViewTouchesBegan(_ pageIndex: Int32, touches: Set<AnyHashable>, with event:
UIEvent) -> Bool {
        return false
    }

    func onPageViewTouchesMoved(_ pageIndex: Int32, touches: Set<AnyHashable>, with event:
UIEvent) -> Bool {
        return false
    }

    func onPageViewTouchesEnded(_ pageIndex: Int32, touches: Set<AnyHashable>, with event:
UIEvent) -> Bool {
        return false
    }

    func onPageViewTouchesCancelled(_ pageIndex: Int32, touches: Set<AnyHashable>, with event:
UIEvent) -> Bool {
        return false
    }
}
```

Note *In the above code, you should specify an existing path to save the image. Here, the path is "/Users/Foxit/Desktop/ScreenCapture.jpg", please replace it with a valid path.*

Step 3: Instantiate a **ScreenCaptureToolHandler** object and then register it to the UIExtensionsManager.

```
var screencaptureToolHandler: ScreenCaptureToolHandler?
...

self.screencaptureToolHandler = ScreenCaptureToolHandler.init(extensionsManager:
self.extensionsManager)
self.extensionsManager.register(self.screencaptureToolHandler)
```

Step 4: Set the **ScreenCaptureToolHandler** object as the current tool handler.

```
self.extensionsManager.setCurrentToolHandler(self.screencaptureToolHandler)
```

Now, we have really finished creating a custom tool in Swift. In order to see what the tool looks like, we need to make it run. Just add the code referred in Step 3 and Step 4 to ViewController.swift.

After finishing all of the above work, build and run the demo. After building the demo successfully, long press and select a rectangular area, and then a message box will be popped up (refer to Figure 4-1). Go to "desktop", we will see the screenshot (refer to Figure 4-2).

This is just an example to show how to create a custom tool in Swift with Foxit MobilePDF SDK. You can refer to it or the demos to develop the tools you want.

5 FAQ

1. What is Bitcode? Does Foxit MobilePDF SDK for iOS support Bitcode?

Bitcode is an intermediate representation of a compiled binary. Including bitcode will allow Apple to re-optimize your app binary in the future without the need to submit a new version of your app to the store.

Yes. Foxit MobilePDF SDK for iOS supports Bitcode since version 3.0.

2. How do I open a PDF document from a specified PDF file path?

Foxit MobilePDF SDK provides multiple interfaces to open a PDF document. You can open a PDF document from a specified PDF file path, or from a memory buffer. For from a specified PDF file path, there are two ways to do that.

The **first** one is that just use the ***openDoc*** interface, which includes the operations of creating a PDF document object (***createFromFilePath***), loading the document content (***load***), and setting the PDF document object to view control (***setDoc***). Following is the sample code:

Note: *The **openDoc** interface is only available for opening a PDF document from a file path. If you want to customize to load a PDF document, you can implement it in the callback function (**FSFileReadCallback**), and then create a document object with a FireRead instance using **createFromHandler**. Next, also load the document content using (**load**), and set the PDF document object to view control using **setDoc**.*

```
#import "ViewController.h"
#import "FoxitRDK/FSPDFViewControl.h"

@interface ViewController : UIViewController

@end

@implementation ViewController
{

    FSPDFViewCtrl* pdfViewCtrl;
}

- (void)viewDidLoad {
    [super viewDidLoad];

    // Get the path of a PDF.
```

```
    NSString* pdfPath = [[NSBundle mainBundle] pathForResource:@"Sample" ofType:@"pdf"];

    // Initilize a FSPDFViewCtrl object with the size of the entire screen.
    pdfViewCtrl = [[FSPDFViewCtrl alloc] initWithFrame: [self.view bounds]];

    // Open an unencrypted PDF document from a specified PDF file path.
    [pdfViewCtrl openDoc:pdfPath password:nil completion:nil];

    // Add the pdfView to the root view.
    [self.view addSubview:pdfViewCtrl];
}
@end
```

The **second** one is that use the ***createFromFilePath*** interface to create a PDF document object, use ***load*** interface to load the document content, and then use ***setDoc*** to set the PDF document object to view control. Following is the sample code:

```
#import "ViewController.h"
#import "FoxitRDK/FSPDFViewControl.h"

@interface ViewController : UIViewController

@end

@implementation ViewController
{

    FSPDFViewCtrl* pdfViewCtrl;
}

- (void)viewDidLoad {
    [super viewDidLoad];

    // Get the path of a PDF.
    NSString* pdfPath = [[NSBundle mainBundle] pathForResource:@"Sample" ofType:@"pdf"];

    // Initialize a PDFDoc object with the path to the PDF file.
    FSPDFDoc* pdfdoc = [FSPDFDoc createFromFilePath:pdfPath];

    // Load the unencrypted document content.
    if(e_errSuccess != [pdfdoc load:nil]) {
        return;
    }

    // Initilize a FSPDFViewCtrl object with the size of the entire screen.
    pdfViewCtrl = [[FSPDFViewCtrl alloc] initWithFrame: [self.view bounds]];

    // Set the document to view control.
    [pdfViewCtrl setDoc:pdfdoc];

    // Add the pdfView to the root view.
    [self.view addSubview:pdfViewCtrl];
}
@end
```

3. What should I do if I want to display a specified page when opening a PDF document?

To display a specified page when opening a PDF file, the interface ***[pdfViewCtrl gotoPage: (int) animated: (BOOL)]*** *is supposed to use.* Foxit MobilePDF SDK utilizes multi-thread to improve rendering speed, so please make sure the document has been loaded successfully before using the ***gotoPage*** interface. There are two ways to realize the feature as follows:

The **first** one is that making a conditional statement in the ***openDoc*** interface to ensure that only when the document loading is complete, then call the ***gotoPage***. If not, the ***gotoPage*** interface will not work, and the first page will be displayed. It is because the ***openDoc*** interface starts a new thread to perform the operation. Following is the sample code:

```
#import "ViewController.h"
#import "FoxitRDK/FSPDFViewControl.h"

@interface ViewController : UIViewController

@end

@implementation ViewController
{

    FSPDFViewCtrl* pdfViewCtrl;
}

- (void)viewDidLoad {
    [super viewDidLoad];

    // Get the path of a PDF.
    NSString* pdfPath = [[NSBundle mainBundle] pathForResource:@"Sample" ofType:@"pdf"];

    // Initilize a FSPDFViewCtrl object with the size of the entire screen.
    pdfViewCtrl = [[FSPDFViewCtrl alloc] initWithFrame: [self.view bounds]];

    // Open an unencrypted PDF document from a specified PDF file path, and go to the
third page when completing the document loading.
    [pdfViewCtrl openDoc:pdfPath password:nil completion:^(enum FS_ERRORCODE error){

        if(error == e_errSuccess)

            // Display the third page.
            [pdfViewCtrl gotoPage:2 animated:NO];
    }];

    // Add the pdfView to the root view.
    [self.view addSubview:pdfViewCtrl];
}
@end
```

The **second** one is that implement the <IDocEventListener> protocol, and then call the ***gotoPage*** interface in the **onDocOpened** event. Following is the sample code:

```
#import "ViewController.h"
#import "FoxitRDK/FSPDFViewControl.h"

@interface ViewController : UIViewController <IDocEventListener>


@end

@implementation ViewController
{
    FSPDFViewCtrl* pdfViewCtrl;
}

- (void)viewDidLoad {
    [super viewDidLoad];

    // Get the path of a PDF
    NSString* pdfPath = [[NSBundle mainBundle] pathForResource:@"Sample" ofType:@"pdf"];

    // Initilize a FSPDFViewCtrl object with the size of the entire screen.
    pdfViewCtrl = [[FSPDFViewCtrl alloc] initWithFrame: [self.view bounds]];

    // Register the PDF document event listener.
    [pdfViewCtrl registerDocEventListener:self];

    // Open an unencrypted PDF document from a specified PDF file path.
    [pdfViewCtrl openDoc:pdfPath password:nil completion:nil];

    // Add the pdfView to the root view.
    [self.view addSubview:pdfViewCtrl];

}

#pragma IDocEventListener

-(void)onDocOpened:(FSPDFDoc *)document error:(int)error
{
    // display the third page.
    [pdfViewCtrl gotoPage:2 animated:NO];
}
@end
```

6 Technical Support

Reporting Problems

Foxit offers 24/7 support for its products and are fully supported by the PDF industry's largest development team of support engineers. If you encounter any technical questions or bug issues when using Foxit MobilePDF SDK, please submit the problem report to the Foxit support team at http://tickets.foxitsoftware.com/create.php. In order to better help you solve the problem, please provide the following information:

- Contact details
- Foxit MobilePDF SDK product and version
- Your Operating System and IDE version
- Detailed description of the problem
- Any other related information, such as log file or error screenshot

Contact Information

You can contact Foxit directly, please use the contact information as follows:

Foxit Support:

- http://www.foxitsoftware.com/support/

Sales Contact:

- Phone: 1-866-680-3668
- Email: sales@foxitsoftware.com

Support & General Contact:

- Phone: 1-866-MYFOXIT or 1-866-693-6948
- Email: support@foxitsoftware.com