DEVELOPER GUIDE

Foxit Plug-in SDK

Microsoft Partner
Gold Independent Software Vendor (ISV)

TABLE OF CONTENTS

Introduction to Plugin Development

Developing Plug-ins provides details to C/C++ developers about plugin development using the Foxit Plug-in SDK. It shows how your plugin can manipulate and enhance both the Foxit PDF Editor and Foxit PDF Reader user interface as well as the contents of underlying PDF documents. This guide also describes how to run samples of Plug-in SDK, provides platform-specific techniques for developing plugins, and lists the Foxit Plug-in SDK header files.

You can use Plug-in SDK to create plugins for Foxit PDF Editor and Foxit PDF Reader. It contains a set of interfaces that let you develop plugins that integrate with Foxit PDF Editor and Foxit PDF Reader and interact with and manipulate PDF documents.

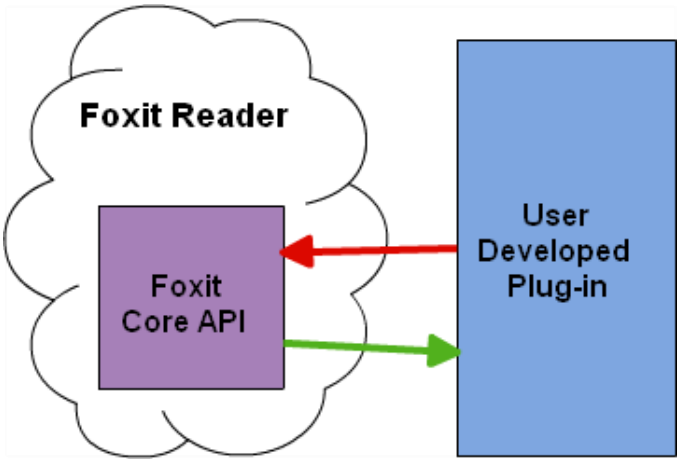
This chapter introduces the Plug-in SDK. Its API descriptions appear in the Foxit Plug-in SDK API Reference.pdf.

About plugins

A plugin is an application that uses the resources of Foxit PDF Editor or Foxit PDF Reader as a host environment. This means that a plugin does not require complex user interface elements. However, it must perform certain basic functions to let Foxit PDF Editor and Foxit PDF Reader know of its presence.

Plugins are dynamically-linked extensions to Foxit PDF Editor and Foxit PDF Reader and are written using the Plug-in SDK API, which is an ANSI C/C++ library. Plugins add custom functionality and are equivalent to dynamically-linked libraries (DLLs) on the Microsoft Windows platform; however, the plugin file name extension is .fpi, not .dll. On Mac OS, the file name extension of a plugin is .dylib .

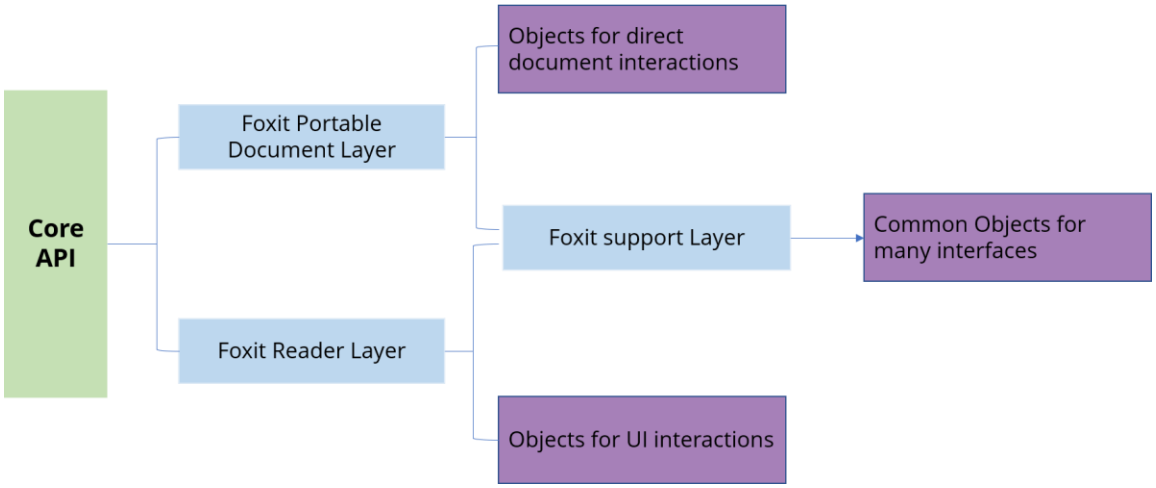
The following diagram shows the relationship between Foxit PDF Reader/Editor and Foxit Plug-in SDK.



About Plug-in SDK

The Plug-in SDK consists of methods that operate on objects located within PDF documents. It is implemented as a standard ANSI C programming library where methods are C functions and objects are opaque data types that encapsulate their data. The Plugin-SDK is supported on Windows (32- and 64-bit) and Mac OS.

The following diagram illustrates the hierarchy of the Plug-in SDK.



Foxit Reader Layer

Foxit Rader Layer enables plugins to control Foxit PDF Editor or Foxit PDF Reader and modify its user interface. Its methods are indicated by the FRD prefix.

Here are some examples of application-level tasks that can be done with the FRD layer

- Add buttons to ribbon toolbar and commands
- Open and close document
- Display simple dialog boxes

Foxit Portable Document Layer

Foxit Portable Document layer provides access to PDF document components such as pages and annotations, which encapsulates many of the PDF objects. Its methods are indicated by the FPD prefix.

Here are some examples of PDF modifications that can be done with the FPD layer,

- Create PDF documents.
- Insert PDF objects into an existing PDF document.
- Add annotations.

Plugins can modify almost all of the data inside a PDF file since the FPD layer provides access to this content.

Foxit Support Layer

Data management is handled by the Foxit Support (FS) layer. The FS layer provides platform-independent data types and methods that support the FRD and FDP layers. The FS layer encapsulates the basic common objects used by the other two layers. Its methods are indicated by the FS prefix.

Objects

All the Plug-in SDK objects are defined as a pointer that represents an internal real object. Objects are obtained by Plug-in SDK methods. Internal objects are opaque so objects' data cannot be directly accessed. Manipulation of objects is achieved by calling corresponding API methods. Objects are passed by reference (vs. passed by value).

Objects names are typically defined in the following structure:

<Layer>_<Name> (Example: *FPD_Document*)

- **Layer**: identifies the API layer (*FPD* = Foxit Portable Document layer)
- **Name**: object's name.

Methods

The name of most Plug-in SDK is typically defined in the following structure:

<Layer><Object><Action><Thing> (example: *FPDDocGetUserPermissions*)

- **Layer**: identifies the API layer (*FPD* = Foxit Portable Document layer)
- **Object**: identifies the object upon which the method acts (*Doc*)
- **Action**: specifies an action that the method performs (*Get*)
- **Thing**: specific to each method. (*UserPermissions*) May not always be present.

Data types

The Plug-in SDK consists of the following data types:

- Scalar
- Simple
- Complex
- Opaque

Scalar types

Scalar (non-pointer) types are based on underlying C language types, but have platform-independent sizes. They are defined in the header file fs_basicExpT.h. All scalar types are Foxit Support Layer types. The following table shows some examples:

Type	Description
FS_BOOL	Boolean
FS_INT32	16-bit unsigned integer
FS_WORD	32-bit unsigned integer
FS_BYTE	Byte (8 bits)
FS_HWND	• QWidget in MacOS • HWND in Windows

Simple types

Simple types represent abstractions or a data structure. The followings are examples of simple data types:

- FS_Rect
- FS_AffineMatrix
- FS_PtrArray
- FS_ByteStringArray

Complex types

Complex types are structures that contain one or more fields. They are used in the following situations:

- To transfer a large number of parameters to or from a method. For example, the FRUIProgressCreate3 method has a parameter which is made up of a complex structure (FR_UIProgressOption).
- To define a data handler or server. For example, your plugin must provide a complex structure populated with callback methods (FR_AppEventCallbacks) when it registers an application event handler.

Opaque types

Many methods hide the concrete C-language representation of data structures. Most methods accept an object and then perform an action on the object. Examples of opaque objects are FPDDoc and FRPageView objects.

Understanding Plugins

About plugin initialization

Plugin Loading and initialization

When Foxit PDF Editor or Foxit PDF Reader is started, it searches the installed plugins and loads them.

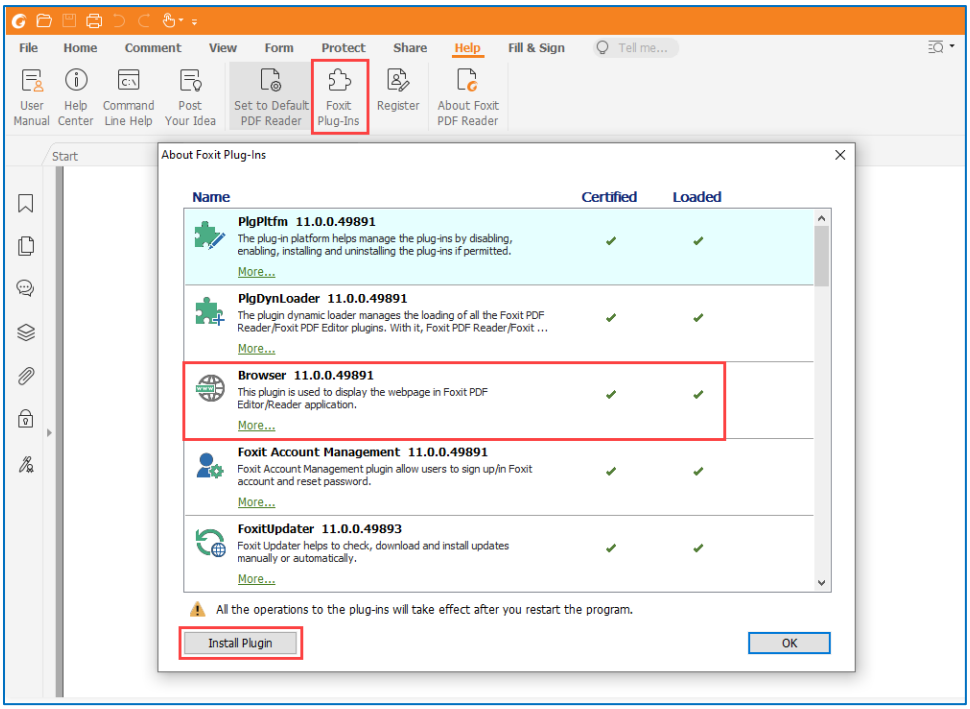
On Windows platform, the plugins will be loaded from specified registry entries (see: Loading from custom directories), and on Mac OS platform, the plugins will be loaded from specified dictionary, such as "~/Library/Application Support/Foxit software/FoxitPDFEditor/(version)/Plugins" or "/Library/Application Support/Foxit software/FoxitPDFEditor/(version)/Plugins".

Install the Plugins

After compiling the plugin successfully, we have to make an installation file for the plugin and then install it to make it loaded when starting the Foxit PDF Editor or Foxit PDF Reader. The installation file is in XML format (e.g., InstallStarter.xml) which is described as below：

```
<?xml version="1.0" encoding="UTF-8"?>
<FoxitPlugin version="1">   // The version of plugin.
<Property
Name="Starter"              // The name of plugin, and it should be unique.
FriendlyName="Starter"      // The display name of plugin.
Description="Starter"       // The description of plugin.
PIPath="Starter.fpi"        // The path of plugin, which is important. Application will load plugin from this path.
LoadBehavior="3"            // The loading behavior of plugin. For third-party development, it can only be set to 3.
MinVersion=""
/>
</FoxitPlugin>
```

You can go to "**Help** -> **Foxit Plug-ins**" to look over the details about all installed Plug-ins. There is a button "**Install Plugin**" in the "**About Foxit Plug-ins**" dialog. Click it to select an installation file to install a Plug-in manually.



Note: If you want to customize the icon displayed in the dialog, you can create a 32*32 PNG icon and make the name same as the Plug-in name. Then put it together with the Plug-in.

Then you must restart Foxit PDF Editor or Foxit PDF Reader for the Plug-ins to take effect.

Note: In Mac OS platform, you need to provide two icons including 16*16 PNG named Plug-in name and 32*32 PNG named Plug-in name2x.

Loading from custom directories

You can also load plugins automatically from custom directories.

- **Windows**

Place the Plug-in in a specified directory and then create registry entries to associate it with Foxit PDF Editor or Foxit PDF Reader. The possible formats of the registry entries are as follows:

For 64-bit Windows, to make your Plug-in available for all Windows users, create a registry key as below:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Foxit Software\Foxit PDF Editor{version}\plugins\Installed\YourPlgName]
```

For 32-bit Windows, to make your Plug-in available for all Windows users, create a registry key as below:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Foxit Software\Foxit PDF
Editor{version}\plugins\Installed\YourPlgName]
```

For 64-bit Windows and 32-bit Windows, to make your Plug-in available for current Windows users, create a registry key as below:

```
[HKEY_CURRENT_USER\Software\Foxit Software\Foxit PDF
Editor{version}\plugins\Installed\YourPlgName]
```

After creating the registry key correctly, set the key-value pair to "YourPlgName" key as below:

```
"FriendlyName "="YourPluginName"
"Description" = "The description of your plugin"
"PIPath "="The full path of your Plug-in"
"LoadBehavior"= “3”
```

- **Mac OS**

On the Mac OS platform, the third-party plugins should be installed in either of these directories:

- ~/Library/Application Support/Foxit software/FoxitPDFEditor/(version)/Plugins
- /Library/Application Support/Foxit software/FoxitPDFEditor/(version)/Plugins

Note: Foxit PDF Editor or Foxit PDF Reader must be restarted in order for the Plug-ins to take effect.

Starting

A plugin must contain a source file that defines the following methods:

Method	Runtime Functionality
PlugInMain / PIMain	Main entry point for the plugin. It is called "PIMain" on Mac OS.
PISetupSDK	Called by the host application to set up the plugin's SDK-provided functionality.
PIHandshake	This routine provides the initial interface between a plugin and the application. It also provides the callback functions to the application that allow it to register the plugin with the application environment.
PIExportHFTs	An extension HFT allows plugins to invoke methods that belong to other plugins. After Foxit PDF Reader/Editor finishes handshaking with all of the plugins, it invokes each plugin's PIExportHFT callback procedure. Extension HFTs are created and added in the PIExportHFT procedure. Once the extension HFT is added to the extension HFT manager, its methods are available to other

	plugins. This callback should only be used for exporting an extension HFT. It should not be used to invoke other Foxit PDF Reader/Editor Core API methods.
PIImportReplaceAndRegister	This function allows you to: Import extension HFTs. Replace functions in existing HFTs. Register to receive notification events.
PIInitUIProcs	This is a callback record contains several initialization methods where plugins execute steps to hook into the user interface of Foxit PDF Reader/Editor. This allows the developer to customize the user interface (e.g., add menu items, toolbar buttons, Ribbon elements, windows, etc.). It is also possible to modify the user interface of Foxit PDF Reader/Editor while running the plugin.
PIInitData	This is the main initialization method where plugins initialize the data only.

The function PluginInMain should be exported from plugin, Foxit PDF Editor or Foxit PDF Reader invokes PluginInMain function when starting loading plugin.

The PluginInMain function is located in the PIMain.cpp file. This source code located in this file is functional and must not be modified.

```
/* The export function of Plug-ins */
__declspec(dllexport) FS_BOOL PlugInMain(FS_INT32* handshakeVersion, PISetupSDKProcType* setupProc)
{
   /*
   ** handshakeVersion indicates which version of the handshake struct that the application has sent to us.
   ** The version you want to use will return to the application, so you can adjust it as desired.
   */


   *handshakeVersion = HANDSHAKE_V0100;
   *setupProc = &PISetupSDK;
   return TRUE;
}
```

The other important method, PISetupSDK, is set in this function, which is called by the host application to set up the Plug-in's SDK-provided functionality.

```
FS_BOOL PISetupSDK(FS_INT32 handshakeVersion, void *sdkData)
{
   if(handshakeVersion != HANDSHAKE_V0100) return FALSE;
   PISDKData_V0100 *pSDKData = (PISDKData_V0100*)sdkData;

   _gpCoreHFTMgr = pSDKData->PISDGetCoreHFT();

   /* Get PID */
   _gPID = pSDKData->PISDGetPID(sdkData);
```

```
    /* Set the Plug-in's handshake routine, which is called next by the host application */
    pSDKData->PISDSetHandshakeProc(sdkData, &PIHandshake);

    /* For compatibility purposes, set the SDK version of the Plug-in,
    so that Foxit Reader will not load the Plug-in whose version is higher than Foxit Reader. */
    pSDKData->PISetSDKVersion(sdkData, SDK_VERSION);

    return TRUE;
}
```

HandShaking

Foxit PDF Editor and Foxit PDF Reader perform a handshake with each plugin as it is opened and loaded. During handshaking, the plugin specifies its name, several initialization procedures, and an optional unload procedure.

A plugin must implement the following handshaking function:

```
FS_BOOL PIHandshake(FS_INT32 handshakeVersion, void *handshakeData)
```

During handshaking, the plugin receives the handshakeData data structure (Type is PIHandshakeData_V0100, which is defined in the fs_piData.h file).

The fs_piData.h header file declares all callback methods that must be located in your plugin. The following shows the function signatures of these callback methods:

- PIHDSetExportHFTsCallback
- PIHDSetImportReplaceAndRegisterCallback
- PIHDSetInitDataCallback
- PIHDSetUnloadCallback

All callbacks return **true** if your plugin's procedure completes successfully or if the callbacks are optional and are not implemented. If your plugin's procedure fails, it returns false. If a plug-in aborts handshaking, the plugin will fail to load. And the application will continue to load another plugin.

To ensure your plugin does not hinder application startup, you must limit code executed in your handshake functions to the minimum.

The following example shows how a plugin's PIHandshake method specifies the plugin callbacks provided during handshake and initialization. The tasks performed by each function is described in the next sections.

```
/** PIHandshake function provides the initial interface between your Plug-in and the application.
    This function provides the callback functions to the application that allow it to
```

```
    register the Plug-in with the application environment.

    Required Plug-in handshaking routine:

    @param handshakeVersion：the version this Plug-in works with.
    @param handshakeData: the data structure used to provide the primary entry points for the Plug-in.
    These entry points are used in registering the Plug-in with the application and allowing the Plug-in to
    register for other Plug-in services.
    @return true to indicate success, false otherwise (the Plug-in will not load).
*/
FS_BOOL PIHandshake(FS_INT32 handshakeVersion, void *handshakeData)
{
    if(handshakeVersion != HANDSHAKE_V0100)
        return FALSE;

    /* Cast handshakeData to the appropriate type */
    PIHandshakeData_V0100* pData = (PIHandshakeData_V0100*)handshakeData;

    /* Set the name and title of plug-in */
    pData->PIHDRegisterPlugin(pData, "Starter", (FS_LPCWSTR)L"Starter");

    /* If you want to export your own HFT, do it in here */
    pData->PIHDSetExportHFTsCallback(pData, &PIExportHFTs);

    /*
    ** If you import Plug-in HFTs, replace functionality, and/or want to register for notifications before
    ** the user has a chance to do anything, do it in here.
    */
    pData->PIHDSetImportReplaceAndRegisterCallback(pData, &PIImportReplaceAndRegister);

    /* Perform your Plug-in's initialization in here */
    pData->PIHDSetInitDataCallback(pData, &PIInit);

    PIInitUIProcs initUIProcs;
    INIT_CALLBACK_STRUCT(&initUIProcs, sizeof(PIInitUIProcs));
    initUIProcs.lStructSize = sizeof(PIInitUIProcs);
    initUIProcs.PILoadMenuBarUI = PILoadMenuBarUI;
    initUIProcs.PIReleaseMenuBarUI = PIReleaseMenuBarUI;
    initUIProcs.PILoadToolBarUI = PILoadToolBarUI;
    initUIProcs.PIReleaseToolBarUI = PIReleaseToolBarUI;
    initUIProcs.PILoadRibbonUI = PILoadRibbonUI;
    initUIProcs.PILoadStatusBarUI = PILoadStatusBarUI;
    pData->PIHDSetInitUICallbacks(pData, &initUIProcs);

    /* Perform any memory freeing or state saving on "quit" in here */
```

```
    pData->PIHDSetUnloadCallback(pData, &PIUnload);

    return TRUE;
}
```

Exporting HFTS

A Host Function Table (HFT) is the mechanism through which plugins invoke methods in Foxit PDF Editor or Foxit PDF Reader, as well as in other plugins. After Foxit PDF Reader/Editor finishes handshaking with all the plugins, it invokes each Plug-in's PIExportHFTs callback procedure.

In the PIExportHFTs procedure, a plugin may export any HFTs it intends to make available to other plugins. This callback should only export an HFT, not invoke other Plug-in SDK API methods. (See "Working with Host Function Tables".)

Note: The only time a plugin can export an HFT is during execution of its PIExportHFTs procedure.

Importing HFTs and registering for notification

After Foxit PDF Editor or Foxit PDF Reader completes invoking each plugin's PIExportHFTs callback method, it invokes each plugin's PIImportReplaceAndRegister callback method. In this method, plugins perform three tasks:

1. Import any special HFTs they use (the standard Foxit Plug-in HFTs are automatically imported). Plugins also may import HFTs any time after this while the plugin is running.
2. Register for notifications. Plugins also may register and unregister for notifications while the plugin is running. A plugin may receive a notification any time after it has registered for it, even if the plugin's initialization callback has not yet been called. This can occur if another plugin initializes first and performs an operation, which causes a notification to be sent. Plugins must be prepared to correctly handle notifications as soon as they register for them.
3. Replace any of the Foxit Plug-In SDK API's replaceable HFT methods.

Note: The only time a plugin may import an HFT or replace a standard API method is within its PIExportHFTs callback procedure. Plugins may register for notifications at this time or any time afterward.

Initialization

After Foxit PDF Editor or Foxit PDF Reader completes calling each plugin's PIImportReplaceAndRegister callback method, it invokes each plugin's PIInit procedure. Plugins can use their initialization procedures to hook into user interface by adding ribbon toolbar buttons, windows, and so on. It is also acceptable to modify user interface later when the plugin is running.

When creating the initialization portion of a plugin, keep the following rules in mind:

- Avoid creating dialog boxes: Do not create a dialog box in your plugin's initialization or do anything else that may interfere with the successful startup of Foxit PDF Editor or Foxit PDF Reader.
- Avoid invoking functions referenced from HFTs exported by other plugins. plugins are not fully initialized until they are invoked or otherwise triggered.
- Avoid invoking system methods that load more system libraries, such as accessing the disk.

Unloading

A plugin's PIUnload procedure should free any memory the plugin allocated and remove any user interface changes it made. Foxit PDF Editor or Foxit PDF Reader invokes this procedure when it terminates or when any of the other handshaking callbacks return false. This function should perform the following tasks:

- Remove and release all user interface elements, HFTs and so on.
- Release any memory or any other allocated resources.

Summarizing a plugin's life cycle

The following steps describe the life cycle of a plugin:

1. At startup, Foxit PDF Editor or Foxit PDF Reader searches for plugin files.
2. For each plugin file, Foxit PDF Editor or Foxit PDF Reader attempts to load the file. If the plugin is successfully loaded, Foxit PDF Editor or Foxit PDF Reader invokes routines in PISetupSDK that complete the handshaking process.
3. Foxit PDF Editor or Foxit PDF Reader invokes callback functions in this order:
    - PIExportHFTs
    - PIReplaceAndRegister

- PIInit

This sequence establishes the linkages between the plugin and Foxit PDF Editor or Foxit PDF Reader, and between the plugin and any other plugins. After all plugins are loaded, Foxit PDF Editor or Foxit PDF Reader continues its own loading and starts the user interface. Then it starts the user session.

Using callback functions

Foxit PDF Editor or Foxit PDF Reader invokes callback functions that you define to perform a specific task. For example, when a user clicks a button located on a toolbar, a callback method is invoked. (See "Creating button callback functions")

Event Notifications

The Foxit Plug-in SDK API provides a notification mechanism so that plugins can synchronize their actions with Foxit PDF Editor or Foxit PDF Reader. Notifications enable a plugin to indicate that it has an interest in a specified event, such as an annotation being modified, and to provide a procedure that Foxit PDF Reader/Editor invokes each time that event occurs. (See "Registering for Event Notifications".)

Using Plugin prefixes

It is important to correctly name all items located in your plugin, such as HFTs, buttons, toolbars, and so on, to ensure they function properly. Failure to do so may cause your plugin to produce unpredictable results when your plugin collides with a plugin of another developer who used the same names.

Using a developer prefix

Every plugin must use the prefix to name its various elements as well as private data it writes into PDF documents. The following sections describe and provide an example of each element that must use a prefix.

Plugin name

ExtensionName, used in plugin handshaking, must use the following syntax: Prefix_PluginName. We recommend the prefix is the company name.

```
/* Set the name as desired*/
pData->PIHDRegisterPlugin(pData, "Foxit_Starter", (FS_LPCWSTR)L"Foxit_Starter");
```

Tool prefixes

Tools names must use the following syntax: Prefix_ToolName. We recommend the prefix is the plugin name, such as Foxit_Starter_HandTool.

Ribbon toolbar and button prefixes

Ribbon toolbar or ribbon button must use the following syntax: Prefix_RibbonElement_ElementName. We recommend the prefix is the plugin name, such as Foxit_Starter_Category_CustomHome or Foxit_Starter_RibbonButton_CustomButton.

For information about creating a toolbar button, see "Ribbon Bar and Buttons".

Creating Plugin

Use the Foxit Plug-in SDK to create plugin applications that interact with PDF documents.

Supported environments

The following table specifies the supported platforms, operating systems, and compilers for Foxit Plug-in SDK development.

Platform	Operating System	Compiler
Windows 32-bit and 64-bit	Windows 7, 8, and 10 (32-bit and 64-bit)	Microsoft Visual Studio 2022 (v143) or later
Mac OS 32-bit and 64-bit	Mac OS Mojave (10.14) or later	Qt 5.12.3

Note: While it may be possible to use the Plug-in SDK in other development environments, such use is not supported. The project files for the sample applications are created and supported only in the listed compiler versions.

Creating a sample

When you start a new plugin, it is recommended that you use the **Starter** sample plugin as a starting point. On Windows, the project file is named Starter.sln and can be found in the "Samples" directory. However, to improve your understanding of creating plugins, the remaining parts of this section discuss what tasks you must perform when creating a plugin from a blank project. When using the **Starter** sample plugin, it is not necessary to perform some of the tasks discussed in this section. For example, you do not need to start a new project, include header files, or add the PIMain source file. However, you still have to add application logic, compile, and build your project.

To create a plugin:

1. Start a new project on Visual Studio or Qt Creator.
2. Include Foxit Plug-in SDK header files.
3. Add the PIMain source file to your project.
4. Add application logic to meet your own requirements.
5. Compile and build your plugin.

Starting a new project

When Foxit PDF Editor or Foxit PDF Reader is started, it searches the installed plugins and load them.

Windows

Visual Studio Project Settings

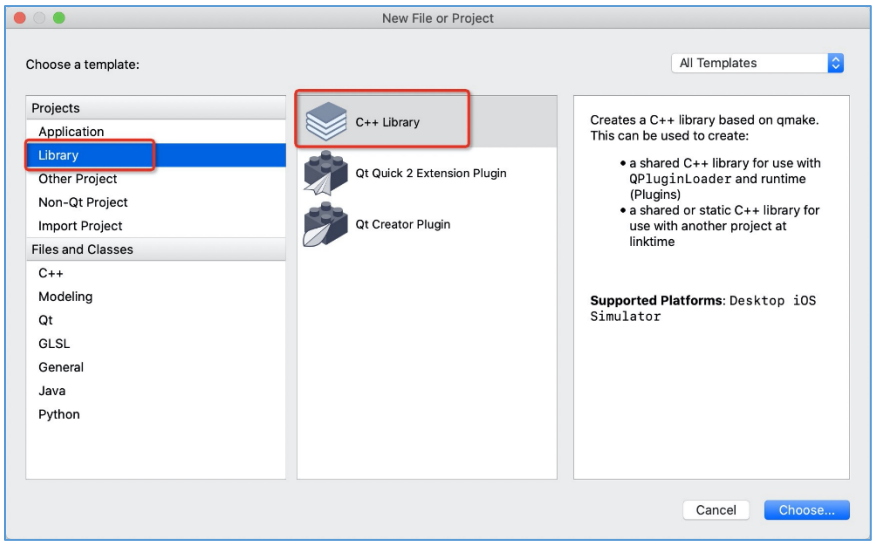
In order to properly develop and debug any Foxit PDF Reader/Editor plugin, Visual Studio must be configured correctly. Below is the demo which shows how to configure Visual Studio 2022 project.

1. Open the **Project Settings** dialog box for the demo project by going to **Project** > **Settings**.
2. Click on the **C/C++** tab and select the **Preprocessor** category.
3. Replace any "_MBCS preprocessor" definition with "UNICODE", "_UNICODE".
4. Click on the **Linker** tab and select the **General** category.
5. In the **Output File Name** text field, enter the full path of your plugin. The plugin path must match the installation directory of Foxit PDF Reader/Editor on your system. The default value is “\Program Files\Foxit Software\Foxit PDF Editor (or Foxit PDF Reader)\plugins”. The extension is “.fpi”.
6. Click on the **Debugging** tab and select the **General** category.
7. In the Executable for debugging session text field, enter the full path of the FoxitPDFReader.exe/FoxitPDFEditor.exe which will load the plugins upon startup. FoxitPDFReader.exe/FoxitPDFEditor.exe must be located at the same directory level as the plugins folder.
8. Click **OK** to apply the new changes and exit the **Project Settings** dialog.
9. Go to **Build** -> **Rebuild Solution**.
10. It is now possible to add breakpoints and debug the plugin project like a normal application. Visual Studio will launch Foxit PDF Reader/Editor when a debugging session begins.

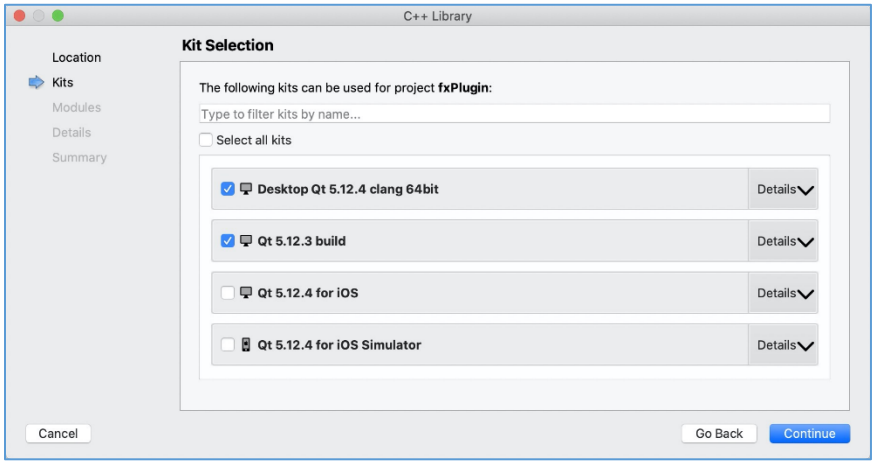
Mac OS

Qt Creator project Settings

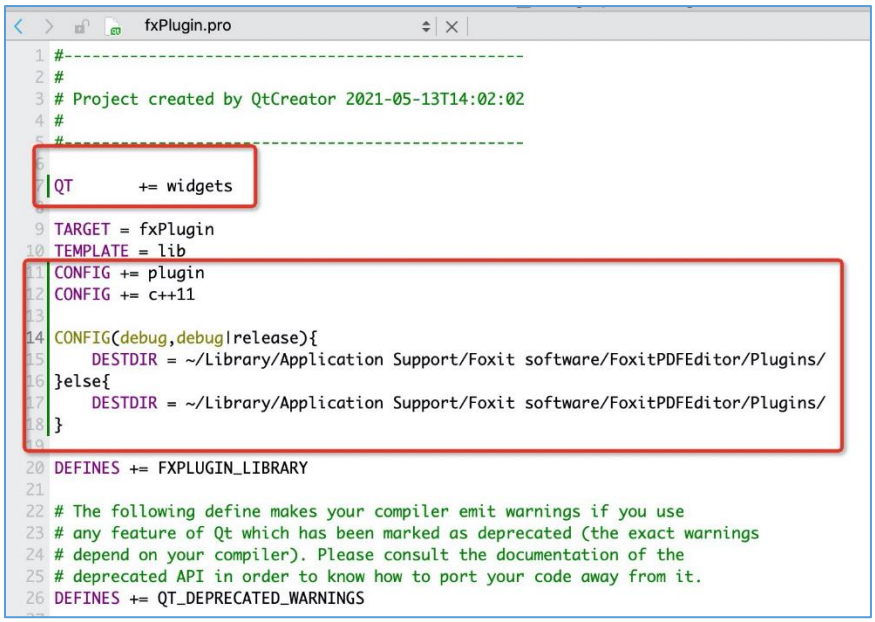
1. Create a new Qt C++ Library base on qmake, and input your project name.



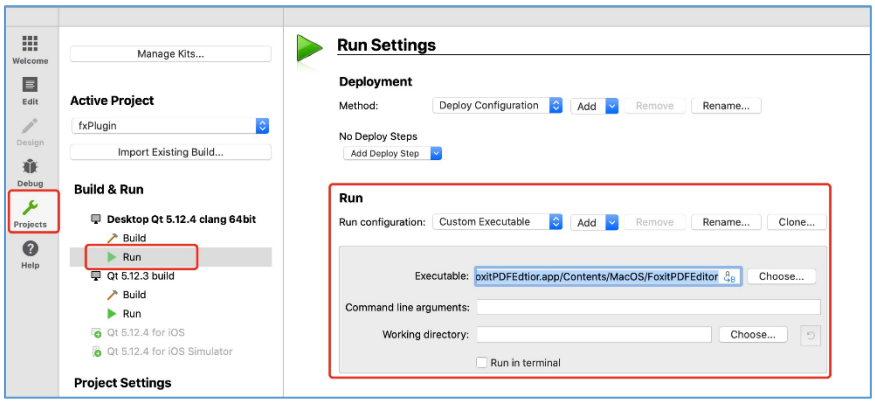
2. Choose Desktop Qt 5.12.4 chang 64-bit (or later) and Qt 5.12.3 build (or later) for kits.



3. Add the configuration Dest DIR information in the Plug-in project configuration file (. Pro), such as [DESTDIR= "~/Library/Application Support/Foxit software/FoxitPDFEditor/Plugins"], which is configured to transfer the generated Plug-ins to the Plug-in loading path of Foxit PDF Reader / Editor.



4. In the Projects configuration, click **Run Setting** to add the Foxit PDF Reader / Editor installation path in the **Executable** text field, for example, [/Applications/FoxitPDFEditor.App/contents/MacOS/FoxitPDFEditor].



5. Run qmaker & Rebuild.

6. It is now possible to add breakpoints and debug the plugin project like a normal application. Qt Creator will launch Foxit PDF Reader/Editor when a debugging session begins.

Including SDK header files

To create a plugin, you must include Plug-in SDK library files, such as header files, into your project. You can link to these library files from within your development environment.

Foxit Plug-in SDK header files must be included in your plugin project. You can find the header files in the following directory:

```
Foxit  Plug-in SDK\PluginSDK\include
```

The following table lists the SDK header files and gives a simple description. In general, including ".basic\fr_callsInclude.h" and ".\basic\fs_pidata.h" header files is enough.

Header file	Description
fr_callsInclude.h	Includes all the xxxCalls.h header files, which define names for referencing Foxit PDF Reader/Editor Plug-in APIs via the corresponding HFTs. Include this file in your plugin.
fr_common.h	Defines Foxit PDF Reader/Editor SDK version, HFT manager which manages referencing Foxit PDF Reader/Editor Plugin SDK APIs.
fs_pidata.h	Defines data structure, types, and other things, which are used to build a handshake routine. This file is shared between Foxit PDF Reader/Editor and plugins.
xxxExpt.h	Contains Types, macros, and structures that are required to use the Host Function Tables.
xxxCalls.h	Defines names for referencing Foxit PDF Reader/Editor Plugin SDK APIs via the corresponding HFTs
xxxTempl.h	Catalogs of functions exported.

Adding the PIMain source file

You must add the PIMain.cpp file to your project in order to create a plugin. This source file contains application logic such as handshaking methods, that are required by plugins. You can find this file in the following directory:

```
Foxit Plug-in SDK\PluginSDK\include\PIMain\
```

Note: As a plugin developer, you will never have to create the application logic that is located in the PIMain.cpp file or modify this file. However, you must include this file in your project.

Adding application logic

You must add a source file to your project that contains the following methods:

- PlugInMain
- PISetupSDK
- PIHandshake
- PIExportHFTs
- PIImportReplaceAndRegister

- PIInitData
- PIUnload

You can copy the source code that is located in the Starter.cpp file (located in the Starter plugin) and paste it. For information about these methods, see "About plugin initialization".

Certifying a Plug-in

Foxit PDF Reader/Editor Plug-ins must be signed with a valid digital certificate after they are compiled and linked in order to be successfully executed by the host Foxit PDF Reader/Editor application. The Foxit PDF Reader/Editor Plug-in SDK provides a signing tool for this purpose. It can be found in the "tools" directory of the evaluation download package.

Note: The license key files **cert.txt** and **frdpisdkey.txt** are trial version and need to be substituted with licensed copies after the evaluation period expires.

Windows

Follow these steps to use the signing tool while developing Plug-ins on Windows. File and folder names can be configured by users. We take Starter Sample for example.

1. Navigate to the Plug-in SDK -> tools directory.
2. Copy cert.txt to Plug-in SDK -> samples -> Starter -> res directory.
3. Open PluginSDK\Samples\Starter\Starter.vcxproj in Visual Studio's Solution Explorer, add cert.txt to the Resources folder.
4. In Visual Studio's Solution Explorer, expand Resource Files, right click on Starter.rc2, and select View Code.
5. Open PluginSDK->tools->dummy.rc in Visual Studio and copy its contents to the code window of Starter.rc2. Save Starter.rc2. Name it depends on your project name. For example, if your project is naming Ribbon, the file name should be Ribbon.rc
6. Build Starter sample Plug-in. From PluginSDK->tools, final build outputs Starter.dll to the ..\..\lib\plugins directory configuration.
7. It is helpful to copy the contents of the application folder (Foxit PDF Reader or Foxit PDF Editor) into the ..\..\lib directory to avoid manually moving the output DLL to the applications plugins folder after each build.
8. Run PIsignatureGen application.
9. The application screen asks for a Plug-in Path, either ..\..\lib\plugins\Starter.fpi, and a Keypair File Path, which takes the private and public key pair found in frdpisdkey.txt file.

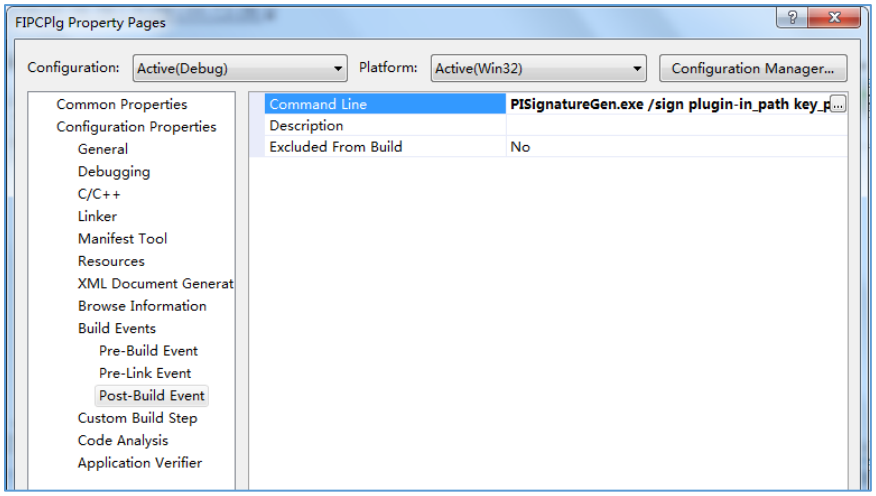
10. Press 'Generate' button, it will sign the Plug-in and leave it in place.
11. It is helpful to leave the PIsignatureGen application running to avoid having to re-enter both fields after each build.
12. The signed Plug-in will now pass the authentication procedure for Foxit applications and start up.

Sign the Plug-in in command line

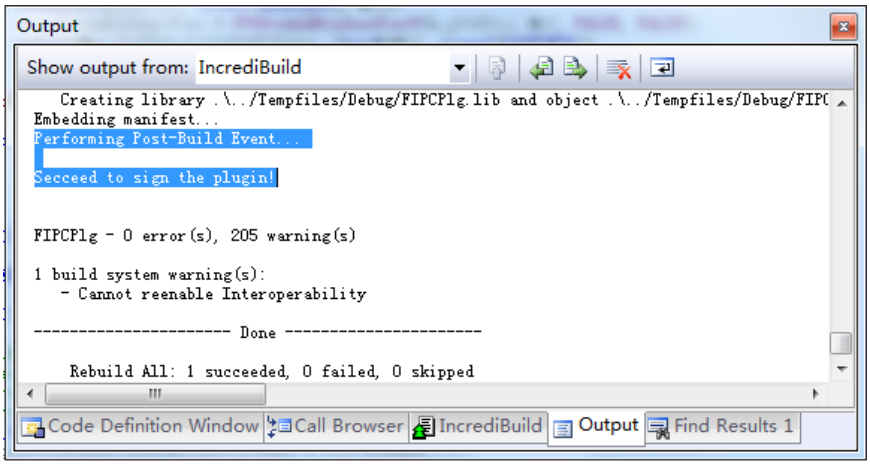
The PIsignatureGen supports command line mode. You can sign the Plug-in as the following command line:

```
PISignatureGen.exe /sign plugin-in_path key_pair_path
```

You can set the command line to Post-Build Event, so that the visual studio can complete the signing process automatically after finishing compiling.



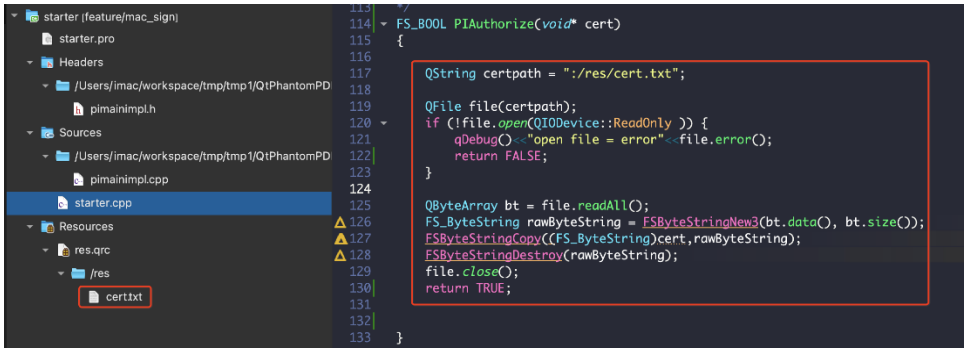
You can see the result in the output window in visual studio after finishing compiling.



Mac OS

Follow these steps to set cert into plugin on Mac OS.

1. Create a resource file in QT project, which can be named "Res".
2. Add Cert.txt file.
3. Implement authorization callback function, Foxit PDF Reader or Foxit PDF Editor will call authorization callback function for reading cert file at appropriate time.



4. Open a terminal, and set the current directory to the directory of the signature tool.
5. Run **PIsignatureGen** tool. The parameter are as follows:

   - --cert -c : the path of cert file
   - --sdkey -k : the path of frdpisdkey.txt
   - --plugin -p : the path of plugin that wanted to be signed.

```
./pisignaturegen \
-c .res /cert.txt \
-k ../../tools /frdpisdkey.txt \
-p ../.. /lib/fxplugins/libstarter.dylib
```

6. If the signature succeeds, then prompt "Succeed to generate", otherwise prompt "Failed to generate".

Note: In the Plug-in directory, a signature file with the same name as the Plug-in name and the extension of dig will be generated. This signature file needs to be in the same directory as the Plug-in.

Applying for a Digital Certificate

Please contact Foxit ISS team or local sales to apply for a digital certificate for certifying plugin. The received digital certificate must be added to the Plug-ins. See Certifying a Plug-in.

Note: If you want to release the Plug-in for Foxit Reader, you will send your Plug-in to Foxit. Foxit Corporation will sign the Plug-in for you and send it back to you.

Working with Documents

This chapter explains how to use the Foxit Plug-in SDK API to perform operations on PDF documents, such as opening a PDF document, creating form control in a PDF document, reloading a PDF document. When working with documents, you use the following types.

FR_Document is a document structure which indicates the view of a PDF document in a window of Foxit PDF Reader/Editor. Usually there is one **FR_Document** object per displayed document, Unlike FPD Document, the **FR_Document** has a window associated with it. The **FPD_Document** may relate to one or more FR_Document objects.

Structure	Description
FR_Document	Primary document structure that represents the view of a PDF document. Here are a few examples of state information that is handled by the PDF document view. • Current page that is on display. • Current zoom level setting. • Enable or disable the Save button depending on whether the contents of a document were modified.

A FPD_Document is the hidden object behind every FR_Document. You can set and retrieve document information fields through FPD_Document objects and make changes to PDF document's contents.

Structure	Description
FPD_Document	Primary document structure that represents the objects that make up a PDF document. Provides access to all of the objects contained within a PDF (e.g., trees of pages, trees of bookmarks, articles, information and security dictionaries, etc.). These objects can be accessed through the FPD layer of the Core API.
FPD_Page	Provides access to a tree of pages.
FPDDocGetPage	Provides access to PDF pages within a document.
FPDDocGetInfo	Provides access to information dictionaries within a document.

Opening PDF document

To open a PDF file in Foxit PDF Editor/Reader, invoke the **FRDocOpenFromFile** method and pass the following arguments:

- A FS_LPCWSTR path that specifies the PDF file to open.
- A FS_LPCSTR password object that specifies the password of the PDF file.
- A FS_BOOL object that specifies whether to show the PDF file in Foxit window.
- A FS_BOOL object that specifies whether to add the PDF file to recent list.

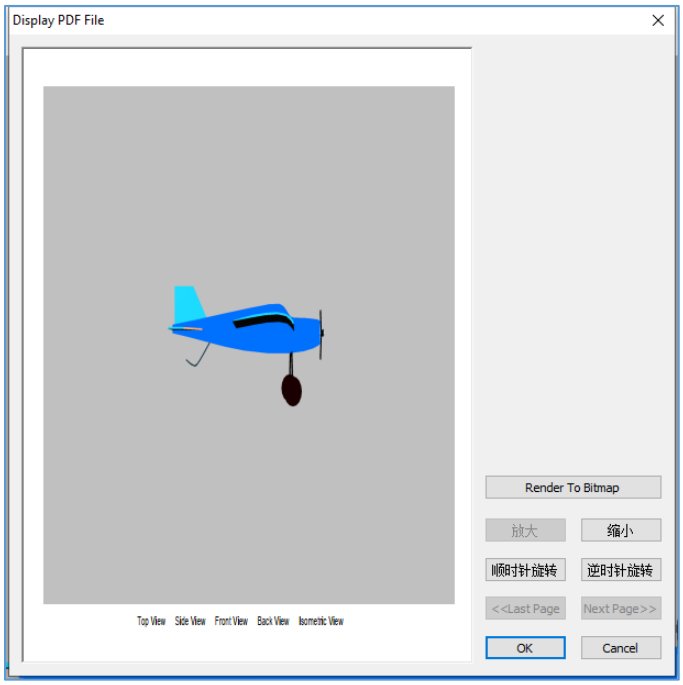
Example: Open a document

```
FS_LPCWSTR openFilePath = L"start.pdf";
FR_Document frDoc = FRDocOpenFromFile(openFilePath,"", TRUE, TRUE);
```

Opening a PDF document in an external window

You can open a PDF document and render it in your own window.

The following image shows a PDF document displayed in an external window. It shows how a PDF document can be manipulated (e.g., zoom in, zoom out, rotate clockwise, and rotate counterclockwise) using the FR_Document structure. The sample also includes a button for rendering a PDF document to a bitmap.



To open a PDF document in an external window, perform the following tasks:

- Create a new window.
- Render PDF page by FPD_RenderDevice.
- Create a FPD_RenderDevice object.
- Load FPD_Page to be render.
- Create FPD_RenderContext object and append page content.
- Render page to window.

Creating a new window

You must create the external windows in which to display a PDF document. To create a window, you can use CDialog in windows platform or QDialog in Mac platform to display external window.

Creating FPD_RenderDevice object

You need to create a new windows device to render the PDF contents. It needs to bind FS_HDC when creating. FS_HDC is an object of HDC in Windows, or a pointer of QPainter in Mac OS.

```
FPD_RenderDevice device = FPDWindowsDeviceNew((void*)devicePainter);
```

Loading FPD_Page to be render

We need to load FPD_Page and parse the contents of the page.

```
FPD_Object dict = FPDDocGetPage(m_pdfDoc, nCurrentPage);
FPD_Page  page = FPDPageNew();
FPDPageLoad(page, m_pdfDoc, dict, TRUE);
/*Parse all the contents of the PDF page.*/
FPDPageParseContent(page, NULL);
```

Setting display appearance

Build a matrix from PDF user space to the targeted device space, according to metrics info: top, left position and page width, height size provided in device space, set rotate of the page display, 0 means no rotate, 45 means rotate to left, 90 means to the bottom, set x-direction and y-direction scale coefficient.

```
FS_AffineMatrix aMatrix = FPDPageGetDisplayMatrix(page, 0, 0, rcDisplay.Width(), rcDisplay.Height(), nRotate);

/*Set the scale value.*/
aMatrix = FSAffineMatrixScale(aMatrix, fScale, fScale);
```

Creating FPD_RenderContext object and append page contents

FPD_RenderContext object is used for rendering a PDF page or a list of page objects. A PDF page can be divided into different layers, including the page content, annotations, and interactive form.

```
FPD_RenderContext  context = FPDRenderContextNew(page, TRUE);
FPDRenderContextAppendPage(context, page, aMatrix);
FPDRenderContextRender(context, device, NULL, NULL);
```

Displaying annotations of the page

```
/*Create the annotation list from the PDF page.*/
FPD_AnnotList annotList = FPDAnnotListNew(page);
/*Display all the annotations to the device. */
FPDAnnotListDisplayAnnots(annotList, page, device, aMatrix, TRUE, NULL);
```

Document permission

You can get and set document permission, it used to check or restrict some functions usage. You can use some enumeration values like below:

Enum type	Descriptions
FR_PERM_MODIFY_CONTENT	modifying page contents or form fields
FR_PERM_EXTRACT_COPY	extracting text or image for copying
FR_PERM_EXTRACT_ACCESS	extracting text or image for accessibility
FR_PERM_ANNOTATE	adding or modifying annotations, filling in forms
FR_PERM_FILL_FORM	Filling in existing form
FR_PERM_PRINT_HIGN	printing in high quality
FR_PERM_ASSEMBLE	Assembling the document
FR_PERM_PRINT	printing

The following example shows how to check if the document has Print permission.

Example: Get permission from document

```
FR_Document frDocument = FRAppGetActiveDocOfPDDoc();
FS_DWORD dPermission = FRDocGetPermissions(frDocument);
if (!(dPermission & FR_PERM_PRINT))
```

```
{
   if (::IsWindowVisible(FRAppGetMainFrameWnd()))
   FRSysShowMessageBox(L"There is no print permission!", MB_OK | MB_ICONINFORMATION, NULL, NULL,
FRAppGetMainFrameWnd());
}
```

The following example shows how to remove the Print permission from document.

Example: Remove permission of documents

```
FS_DWORD aPermissions = dStyle & ~FR_PERM_PRINT;
FRDocSetPermissions(frDocument, aPermissions);
```

Organizing pages

Based on Foxit Plug-in SDK API, you can replace pages, extract specified pages, and insert pages to specified files.

Replacing pages

You can replace pages that is based on FRDocReplacePages, you can perform the following tasks:

- Get FR_Document object of file to be replaced.
- Open and get FPD_Document object of the replaced file.
- Set the number of pages to be replaced.
- Call FRDocReplacePages by passing above object.

Example: Replace pages

```
FR_Document frDocument = FRAppGetActiveDocOfPDDoc();
// Open and parse the replaced file.
FS_LPCWSTR inputfile = L"replace.pdf";
FR_Document frsrcDocument = FRDocOpenFromFile(inputfile, (FS_LPCSTR)L"", false, false);
FPD_Document fpdDocument = FRDocGetPDDoc(frsrcDocument);
// Set the number of pages to be replaced.
FS_WordArray arr = FSWordArrayNew();
FSWordArrayAdd(arr, 0);
// The second parameter is that the original file that needs to be replaced from the first page.
FS_BOOL bRet = FRDocReplacePages(frDocument, 0, fpdDocument, arr);
```

Extracting pages

You can extract pages that is based on FRDocExtractPages, you can perform the following tasks:

- Get FR_Document object of source file that needs to extract.
- Create new FPD_Document for extract pages.
- Set the number of pages to be extracted.
- Call FRDocExtractPages by passing above object.
- Save the new document.

Example: Extract pages

```
FR_Document frDocument = FRAppGetActiveDocOfPDDoc();
// Create a memory document.
FPD_Document fpdDoc = FPDDocNew();
// Set the number of pages to be extracted.
FS_WordArray arr = FSWordArrayNew();
FSWordArrayAdd(arr, 0);
FS_BOOL bExtract = FRDocExtractPages(frDocument, arr, fpdDoc);
if (bExtract)
{
    // Save the memory file after being extracted.
  FS_LPCWSTR inputfile = L"extract.pdf";
  FS_BOOL bSave = FPDDocSave2(fpdDoc, inputfile, 0, FALSE);
}
```

Inserting pages

You can insert pages that is based on FRDocInsertPages, you can perform the following tasks:

- Get FR_Document object of source file that needs to insert pages.
- Open and get FPD_Document object of the specified file to be insert.
- Set the page number of the file to be inserted.
- Call FRDocInsertPages by passing above object.

Example: Insert pages

```
FR_Document frDocument = FRAppGetActiveDocOfPDDoc();
// Open the file to be inserted.
FS_LPCWSTR inputfile = L"insert.pdf";
FR_Document frsrcDocument = FRDocOpenFromFile(inputfile, (FS_LPCSTR)L"", false, false);
FPD_Document fpdDocument = FRDocGetPDDoc(frsrcDocument);
```

```
// Set the page number of the file to be inserted.
FS_WordArray arr = FSWordArrayNew();
FSWordArrayAdd(arr, 0);
FS_WideString wsName = FSWideStringNew3(L"InsertDoc", -1);
// Set the corresponding page to insert the file from the specified position.
FS_BOOL bRet = FRDocInsertPages(frDocument, 0, fpdDocument, arr, TRUE, FALSE, FALSE, wsName, TRUE);
```

Converting PDF document

Based on Foxit Plug-in SDK API, you can convert PDF document to other file type. The supported types are as bellow:

Type	File Extension
HTML	HTML (*.htm,*.html)|*.htm;*.html
Docx	Word Document (*.docx)|*.docx
PNG	PNG (*.png)|*.png
xlsx	Excel (*.xlsx)|*.xlsx
Rich Text	Rich Text Format (*.rtf)|*.rtf
TXT Files	TXT Files (*.txt)|*.txt
PowerPoint	PowerPoint (*.pptx)|*.pptx
JPEG2000	JPEG2000 (*.jpf,*.jpx,*.jp2,*.j2k,*.jpc)|*.jpf;*.jpx;*.jp2;*.j2k;*.jpc
BMP	BMP (*.bmp,*.dib,*.rle)|*.bmp;*.dib;*.rle
XPS Document	XPS Document(*.xps,*.oxps)|*.xps;*.oxps
TIFF	TIFF (*.tiff,*.tif)|*.tiff;*.tif

Calling FRDocConvertPdfToOtherFormat2, you can convert current document to a specified file with passing the following parameters:

- wsDesPath: The file path after finishing converting.
- szFileExt: The file extension that wants to be converted to.

Example: Convert PDF document

```
FS_LPCWSTR  convertFilePath = L"convert.html";
FS_LPCSTR convertType = "HTML (*.htm,*.html)|*.htm;*.html";
bRet = FRDocConvertPdfToOtherFormat2(convertFilePath, convertType);

FS_LPCWSTR  convertFilePath = L"convert.docx";
```

```
FS_LPCSTR convertType = "Word Document (*.docx)|*.docx";
bRet = FRDocConvertPdfToOtherFormat2(convertFilePath, convertType);
```

Saving documents

You can save documents using one of the following methods:

FRDocDoSaveAs

The FRDocDoSaveAs method to save as the document will display file dialog.

Example:

```
FRDocDoSaveAs(frDocument);
```

FRDocDoSave2

The FRDocDoSave2 method saves the current document with passing the following arguments:

- A FR_Document object that represents the document that needs to be saved.
- A FR_DocSaveProc object that represents the callback when document has been saved completely.
- A pProcData object that represents the client data passed to FR_DocSaveProc.
- A FS_BOOL object that represents whether to show progress bar.
- A FS_BOOL object that represents whether to optimize the PDF document.

Example:

```
FRDocDoSave2(frDocument, NULL, NULL, TRUE, TRUE);
```

FRDocDoSaveAs3

The FRDocDoSaveAs3 method saves the document to a specified path with passing the following arguments:

- A FR_Document object that represents the document that needs to be saved as.
- A FS_LPCWSTR object that represents the file path to be saved as. If it is empty, the method will return FALSE.
- A FR_DocSaveProc object that represents the callback when document has been saved completely.
- A pProcData object that represents the clientdata passed to FR_DocSaveProc.

- A FR_SaveDocOption object that represents save option like whether to show progress bar or to optimize PDF file.

Example:

```
FR_SaveDocOption option;
// If you need to optimize document, it will clear all cached appearance, when the application changed any appearance settings.
option.bDocPDFOptimizer = TRUE;
// Whether to show prompt after finishing saving.
option.bPromptInfo = TRUE;
// Whether to save as a temp file.
option.bSaveAsTempFile = FALSE;
// Whether to show progress bar.
option.bShowProgressBar = TRUE;
FRDocDoSaveAs3(frDocument, L"\\saveasTemp.pdf", NULL, NULL, option);
```

Closing document

You can use the FRDocClose to close documents with passing the following arguments:

- A FR_Document object that represents the document that needs to be closed.
- A FS_BOOL object that represents whether to prompt user when the file changed.
- A FS_BOOL object that represents whether to delay close.
- A FS_BOOL object that represents whether to show cancel button in progress bar.

Example:

```
FR_Document frDocTemp = FRDocFromPDDoc(fpdDoc);
bool isClosed = FRDocClose(frDocument, FALSE, TRUE, FALSE);
```

Working with Document Views and Page Views

This chapter explains how to display the document views and modify the contents. In Foxit PDF Reader/Editor, there are mainly two object types for document view, one is **FR_DocView** object type, the other is **FR_PageView** type. The **FR_PageView** object is the area of the Foxit PDF Reader/Editor window that displays the visible content of a document page. The **FR_DocView** object is to centrally manage **FR_PageView**, and you can obtain specific **FR_PageView** objects or perform certain operations on them through **FR_DocView**.

About page coordinates

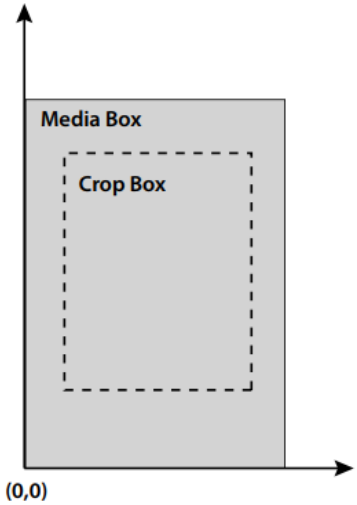
When working with page views and page contents, most times it is necessary to specify page coordinates.

Two coordinate systems are applicable to the Foxit Plug-in SDK API:

- User space
- Device space

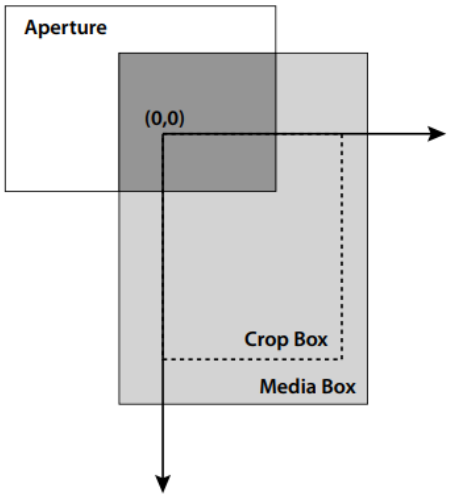
User space is the coordinate system used within PDF documents. It specifies coordinates for most document objects.

The following diagram shows a user space coordinate system.



Device space specifies coordinates in screen pixels and is used to specify screen coordinates of windows.

The following diagram shows a device space coordinate system.



The FRPageViewRectToDevice method can transform a rectangle's coordinates from user space to device space. For example, you can get a user space coordinates of a rectangle. However, to display an outline around the rectangle, you must convert user space coordinates to device space coordinates.

About Document views

The document view is managed as the **FR_DocView** object obtained through the FRDocGetDocView method, which requires the following parameters:

- A **FR_Document** object, which represents the PDF document containing the page on which the page view is based. (See "Opening PDF Documents" in the chapter "Working with Documents".)
- Specific document view management object index. The total number of document view management objects can be obtained through the FRDocCountDocViews method. The index value cannot exceed the total number obtained. The index value starts from 0, that is, when the index value is 0, it represents the first document view management object.

You can perform related view management operations through the related methods of **FR_DocView** object, for example:

- Set the zoom type and zoom ratio of the document view
- Scroll the document view to the specified display position
- Redraw the document view
- Get a specific page view object
- Jump to a specific page view

The following code example shows a page view based on a **FR_Document** object named hDocument. The specified page of the page view index is 5 (page 6 is displayed).

Example: Set the zoom type and zoom ratio

```
FS_INT32 numDocView = FRDocCountDocViews(hDocument);
if(numDocView > 0)
{
		FR_DocView hDocView = FRDocGetDocView(hDocument, 0);
		FRDocViewZoomTo(hDocView, FR_ZOOM_MODE_ACTUAL_SCALE, 2.5);
}
```

About Page views

The document page view is the **FR_PageView** object obtained through the FRDocViewGetPageView method, which requires the following parameters:

- A FR_DocView object, which represents the page view management object in the PDF document. (See: About Document views)
- Specific page view index. The total number of document page views can be obtained through the FRDocViewCountPageViews method. The index value cannot exceed the total number obtained. The index value starts from 0, that is, when the index value is 0, it represents the first page view object.

You can perform related page view operations through the related methods of **FR_PageView** object, for example:

- Convert the user space coordinates and device space coordinates on the page view.
- Get the page view matrix coordinates.
- Add, delete, and retrieve page view annotations.
- update the page view annotations.

The following code example shows a page view based on an **FR_DocView** object named hDocView. The specified page of the page view index is 5 (page 6 is displayed).

Example: Page coordinates user space and device space interchange

```
FS_INT32 indexPageView = 5;
FS_INT32 numPageView = FRDocViewCountPageViews(hDocView);
if(indexPageView < numPageView)
{
		FR_PageView hPageView = FRDocViewGetPageView(hDocView, indexPageView);

		FS_Rect rect;
		rect.bottom = 200;
		rect.left = 100;
		rect.right = 200;
		rect.top = 100;

		FS_FloatRect outRect;
		FRPageViewDeviceRectToPage(hPageView, &rect, &outRect);

		FS_Rect rect1;
		FRPageViewRectToDevice(frPageView, &outRect, &rect1);
}
```

Example: Display a page view

```
FS_INT32 indexPageView = 5;

FS_INT32 numDocView = FRDocCountDocViews(hDocument);
if(numDocView > 0)
{
		FR_DocView hDocView = FRDocGetDocView(hDocument, 0);

		FS_INT32 numPageView = FRDocViewCountPageViews(hDocView);
		if(indexPageView < numPageView)
  {
				FRDocViewGotoPageView(hDocView, indexPageView);
				FRDocViewDrawNow(hDocView);
		}
}
```

Inserting Text into PDF Documents

This chapter explains how to use FPD Layer funcitons to insert text to the document. Foxit Portable Document layer provides access to PDF document components such as pages and annotations, it is not used to modify the user interface of Foxit Editor PDF. We will introduce detailed steps to show how to start for creating a new PDF document. You can follow the steps below:

1. Create a new PDF document
2. Create a new page
3. Create a text object
4. Create a textstate object of text
5. Create colorstate object of text
6. Inset text to page object
7. Refresh page content stream
8. Save the PDF document

Creating a new PDF document

You can create a new PDF document by invoking the **FPDDocNew** method. This method returns a **FPD_Document** object that only contains document catalog. The following code example creates a **FPD_Document** object by invoking **FPDDocNew** method.

Shows a sample catalog object (See PDF Reference 3.6.1).

```
1 0 obj
   << /Type /Catalog
      /Pages 2 0 R
      /PageMode /UseOutlines
     /Outlines 3 0 R
  >>
endobj
```

Example: Create a new PDF document

```
// Create a new document.
```

```
FPD_Document pPDFDoc = FPDDocNew();
```

Creating a new page

You can create a new page by invoking the FPDDocCreateNewPage method, this method will return a **FPD_Object** object that is page dictionary (see PDF Reference 3.2.6). Then you need to create a **FS_FloatRect** object, which represents a rectangle region that specifies the page size. After declaring **FS_FloatRect** object, you can specify the left, top, right and bottom attributes. Last, you need to set MediaBox attribute, you can set by FPDDictionarySetAtRect method.

Note: MediaBox (see PDF Reference 3.6.2). A rectangle expressed in default user space units (see About page coordinates), defining the boundaries of the physical medium on which the page is intended to be displayed or printed. That is a required attribute.

Example: Creating a new page

```
// Create a new page.
FPD_Object pPageDict = FPDDocCreateNewPage(pPDFDoc,0);
if (!pPageDict) return;

// Set page rect.
FS_FloatRect pageRect;
pageRect.left = 0;
pageRect.bottom = 0;
pageRect.right = 612;
pageRect.top = 792;
FPDDictionarySetAtRect(pPageDict, "MediaBox", pageRect);
```

Creating font object

You can create text objects by invoking the FPDTextObjectNew method, this method will return a **FPD_PageObject** object.

Creating CJK font object

If you want to insert text of CJK font, you need create specified CJK font first like “SimSun”.

- First, use system function of CreateFont to create a HFONT;
- Second, construct **LOGFONTW** object by get the information from HFONT;
- Then, use **FPDDocAddWindowsFontW** to create **FPD_Font** object and add to doc.

Example: Creating CJK font object

```
//create font
HFONT hFont = ::CreateFont(12, 0, 0, 0, FW_NORMAL, FALSE, 0, 0, GB2312_CHARSET,
      OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
      DEFAULT_PITCH | FF_DONTCARE, L"SimSun");
LOGFONTW lf;
memset(&lf, 0, sizeof(LOGFONTW));
::GetObject(hFont, sizeof(LOGFONTW), &lf);
::DeleteObject(hFont);
//Add font to doc
FPD_Font font = FPDDocAddWindowsFontW(pPDFDoc, &lf, FALSE, FALSE);
```

Creating a text object

You can create text objects by invoking the FPDTextObjectNew method, this method will return a **FPD_PageObject** object.

We can set text info by FPD_PageObject object:

- FPDTextObjectSetPosition: set text position.
- FPDTextObjectSetText: set text.
- FPDTextObjectSetTextState: set font size and font.
- FPDPageObjectSetColorState: set text color.

We will describe TextState and ColorState later, which need to create a new object.

Example: Creating a text object

```
// Create a new text object.
FPD_PageObject textObj = FPDTextObjectNew();
FS_ByteString bsText = FSByteStringNew3("Hello Word!", -1);
// Set position.
FPDTextObjectSetPosition(textObj, 200, 400);
FPDTextObjectSetText(textObj, bsText);
// Set text content.
FSByteStringDestroy(bsText);
```

Creating a textstate object

You can create textsatate by invoking the FPDTextStateNew method, this method will return a **FPD_TextState** object.

For **FPD_TextState** object, you can set Font, FontSize, Matrix, CharSpace, WordSpace, and mainly use:

- FPDTextStateSetFont
- FPDTextStateSetFontSize

Construct a FPD_Font by FPDDocAddStandardFont, you can pass the follow arguments:

- A FPD_Document object that represents the PDF document.
- A string object that represents font name.
- A FPD_FontEncoding object that represents font encoding.

Example: Creating FPD_TextState object

```
FPD_TextState textState = FPDTextStateNew();
FPD_Font font = FPDDocAddStandardFont(pPDFDoc, "Times-Bold", NULL);
FPDTextStateSetFont(textState, font);
FPDTextStateSetFontSize(textState, 25);
FPDTextObjectSetTextState(textObj, textState);
FS_FLOAT  matrix[4]{ 1, 0, 0, 1 };
FPDTextStateSetMatrix(textState, matrix);
FPDTextStateDestroy(textState);
```

Creating a colorstate object

You can create colorstate by invoking the FPDColorStateNew method, this method will return a **FPD_ColorState** object.

For FPD_ColorState object, we can set Font color, we mainly use FPDColorStateSetFillColor.

Example: Creating FPD_ColorState object

```
FPD_ColorState pColorState = FPDColorStateNew();
if (FPD_Color pFillColor = FPDColorStateGetFillColor(pColorState))
{
        FPDColorSetColorSpace(pFillColor, FPDColorSpaceGetStockCS(FPD_CS_DEVICERGB));
}
if (FPD_Color pStrokeColor = FPDColorStateGetStrokeColor(pColorState))
{
        FPDColorSetColorSpace(pStrokeColor, FPDColorSpaceGetStockCS(FPD_CS_DEVICERGB));
}
FS_FLOAT rgb[3];
rgb[0] = 1.0f;
rgb[1] = 0.0f;
rgb[2] = 0.0f;
FPDColorStateSetFillColor(pColorState, FPDColorSpaceGetStockCS(FPD_CS_DEVICERGB), rgb, 3);
FPDPageObjectSetColorState(textObj, pColorState);
FPDColorStateDestroy(pColorState);
```

Inserting text to page object

Last, to insert text object to page, you can get FPD_Page from loading document by page dictionary, and get position for inserting text object, and then call FPDPageInsertObject to insert.

You can follow the steps as below:

1. Create a FPD_Page object.
2. Construct a page by FPDPageLoad, you can pass the follow arguments:
    - A FPD_Page object that represents the new page.
    - A FPD_Document object that represents the PDF document.
    - A FPD_Object object that represents the page dictionary.
    - A bool object that represents Whether images and masks used in page rendering will be cached or not.
3. Get the position to insert text.
4. Insert text object by FPDPageInsertObject.

Example: Insert text to page object

```
FPD_Page fpdPage = FPDPageNew();
FPDPageLoad(fpdPage, pPDFDoc, pPageDict, TRUE);
if (fpdPage)
{
		FS_POSITION pos = FPDPageGetLastObjectPosition(fpdPage);
		// Add text to page.
		FPDPageInsertObject(fpdPage, pos, textObj);
}
```

Insert CJK text to page object

If you want insert CJK text, when after create specified CJK font, it needs to get char code of the specified text. We can use FPDFontCharCodeFromUnicode to get char code by specified FPD_Font object.

- Use FPDFontCharCodeFromUnicode to get char code from text by specified font.
- Use FPDTextObjectSetText3 to set text by char codes.

Example: Insert CJK text

```
FS_ByteString bsText = FSByteStringNew3("你好，欢迎来到 PDF 世界!", -1);
//Set position
FPDTextObjectSetPosition(textObj, 200, 400);

FS_LPCWSTR content = L"你好，欢迎来到 PDF 世界!";
int len = lstrlen(content);
FS_DWORD* pCharCodes = new FS_DWORD[len + 1];
FS_FLOAT* pKern = new FS_FLOAT[len + 1];

for (int i = 0; i < len; i++)
{
   pCharCodes[i] = FPDFontCharCodeFromUnicode(font, content[i]);
   if ((pCharCodes[i] == 0xFFFFFFFF) || (pCharCodes[i] == 0))
   {
      pCharCodes[i] = ' ';
   }
   pKern[i] = 0;
}
pCharCodes[len] = 0;

FPDTextObjectSetText3(textObj, len, pCharCodes, pKern);
//set text content
FSByteStringDestroy(bsText);
```

Refreshing page content stream

After adding object, you need to call PFDPageGenerateContent for refreshing page content stream.

Example: Refresh page content

```
FPDPageGenerateContent(fpdpage);
```

Saving documents

You can save the document by invoking FPDDocSave to save it to a specified file path, you can pass the follow arguments:

- A FPD_Document object that represents the PDF document.
- A string object that represents the path where the file is saved to.
- A createflags object that specifies flags for PDF Creator.
- A bool object that specifies whether to set data compression.

The following code example saves the PDF document to a specified directory as new.pdf.

Example: Save document

```
FPDDocSave(pPDFDoc, ".\new.pdf", FPDFCREATE_OBJECTSTREAM, TRUE);
```

When you open the file, it will look like as bellow:

Working with Annotations

This chapter explains how to create new annotations, modify annotations, delete annotations, and add annotations to markup panel. An annotation associates an object such as a note, sound, or movie with a location on a page of a PDF document, or provides a way to interact with the user by means of the mouse and keyboard. PDF includes a wide variety of standard annotation types. See PDF Reference 8.4.

About annotations

The Foxit Plug-in SDK API provides methods for working with annotations in PDF documents. Annotations are represented by a **FPD_Annot** in FPD Layer or **FR_Annot** in FR Layer typedef.

Several annotation types exist, which are identified by their subtype. Each subtype can have additional properties that extend the basic ones. The subtype for Highlight is Highlight. The subtype for link annotations is link.

You can use FPDDictionary methods to set and get annotations properties, such as 'location', 'color', 'subtype'.

Working with Highlight annotations

The Foxit Plug-in SDK API allows you to create markup annotations, retrieve and modify attributes of an existing annotation. We take Highlight annotation as an example to introduce how to create a new annotation, or modify/delete a specified annotation.

Creating Highlight annotations

You can create highlight annotations by performing the follow tasks:

- Create a **FPD_Object** object, which is a dictionary object, and it is used to set annot dictionary. (See: PDF Reference 1.7 – 8.4.1 Annotation dictionaries)
- Set subtype attribute to FPD_Object. You can use FPDDictionaryAddValue to add key and value to **FPD_Object**.
- Set QuadPoints attribute for location to FPD_Object. An array of 8 × n numbers specifying the coordinates of n quadrilaterals in default user space. Each quadrilateral encompasses a

word or group of contiguous words in the text underlying the annotation. The coordinates for each quadrilateral are given in the order x1 y1 x2 y2 x3 y3 x4 y4 specifying the quadrilateral's four vertices in counterclockwise order. The text is oriented with respect to the edge connecting points (x1, y1) and (x2, y2).

- Set C attribute of the annotation, it represents the color of the annot. An array of numbers in the range 0.0 to 1.0, representing a color used for the following purposes:
    - The background of the annotation's icon when closed
    - The title bar of the annotation's pop-up window
    - The border of a link annotation
- Set the **rect** attribute of the annotation, it defines the location of the annotation on the page in default user space units.
- Set contents attribute of the annotation, it to be displayed for the annotation or, if this type of annotation does not display text, an alternate description of the annotation's contents in human-readable form.
- After adding markup annotations, you can call FRMarkupPanelAddAnnot to add annotations to comments panel, otherwise the comments panel will not show the annotations.

Example: Create highlight annotations

```
FR_DocView frDocView = FRDocGetCurrentDocView(frDocument);
FR_PageView frPageView = FRDocViewGetCurrentPageView(frDocView);
FPD_Object fpdObject = FPDDictionaryNew();
FS_ByteString strtype = FSByteStringNew3("Annot", -1);
FPD_Object fpdfstringAnnot = FPDStringNew(strtype, 0);
FPDDictionaryAddValue(fpdObject, "Type", fpdfstringAnnot);

FS_ByteString strSubtype = FSByteStringNew3("Highlight", -1);
FPD_Object fpdfstringSubtype = FPDStringNew(strSubtype, 0);
FPDDictionaryAddValue(fpdObject, "Subtype", fpdfstringSubtype);
FPDDictionaryAddValue(fpdObject, "Subj", FPDStringNew(FSByteStringNew3("Highlight", -1), 0));

FS_FloatRect rectFirst;
rectFirst.left = 100;
rectFirst.right = 300;
rectFirst.top = 300;
rectFirst.bottom = 100;

FPD_Object quad = FPDArrayNew(); // {x1, y1, x2, y2, x3, y3, x4, y5}
FPDArrayAddNumber(quad, rectFirst.left); // x1
FPDArrayAddNumber(quad, rectFirst.top); // y1
FPDArrayAddNumber(quad, rectFirst.right); // x2
FPDArrayAddNumber(quad, rectFirst.top); // y2
FPDArrayAddNumber(quad, rectFirst.left); // x3
```

```
FPDArrayAddNumber(quad, rectFirst.bottom); // y3
FPDArrayAddNumber(quad, rectFirst.right); // x4
FPDArrayAddNumber(quad, rectFirst.bottom); // y4
FPDDictionaryAddValue(fpdObject, "QuadPoints", quad);

FPD_Object quadColor = FPDArrayNew();
FPDArrayAddNumber(quadColor, 1); // 1 means RGB Color
FPDArrayAddNumber(quadColor, 0.929412); // Defines the color
FPDArrayAddNumber(quadColor, 0); // Defines the color
FPDDictionaryAddValue(fpdObject, "C", quadColor);

FS_LPCWSTR ws = (FS_LPCWSTR)L"This is initial text";
FPD_Object fpdfstringPopout = FPDStringNewW(ws);
FPDDictionaryAddValue(fpdObject, "Contents", fpdfstringPopout);
FPDDictionarySetAtRect(fpdObject, "Rect", rectFirst);

FR_Annot frAnnot = FRPageViewAddAnnot(frPageView, fpdObject, 0);
FR_MarkupPanel panel = FRMarkupPanelGetMarkupPanel();
FRMarkupPanelAddAnnot(panel, frAnnot, TRUE, TRUE);
FRDocReloadPage(frDocument, 0, FALSE);
```

Modifying specified type annotations

You can modify an annotation after you get it. For example, you can retrieve an existing highlight annotation and modify its content.

Before you modify an annotation, determine whether the annotation is the match subtype. That is, before modify a highlight annotation, ensure that the annotation is a highlight annotation. You can determine whether an annotation is the correct subtype by invoking the FRAnnotGetType method. This method requires a FR_Annot object and return **FS_ByteString** object that specifies the annotation's subtype.

When modifying a highlight annotation's content, it is recommended that you check its contents. For example, you can retrieve all annotations in the page, check the type of the annotation, then invoke the FPDDictionarySetAtString to modify the contents to annot dictionary.

The following code example iterates through all annotations located in the current page. Each valid annotation is checked to determine whether it is a highlight annotation. This task is performed by invoking the FRAnnotGetType method. If the annotation is a highlight annotation, get annot dictionary by FPDAnnotGetAnnotDict, and then set contents to the dictionary.

Last the same is needed to use FRMarkupPanelRefreshAnnot to refresh annotation's modification.

Example: Create highlight annotations

```
for (int i = 0; i < count; i++)
```

```
{
		FR_Annot annot = FRPageViewGetAnnotByIndex(frPageView, i);
		FS_ByteString type = FSByteStringNew();
		FRAnnotGetType(annot, &type);
		FS_ByteString bsHighlightType = FSByteStringNew3("Highlight", -1);
		FSByteStringDestroy(type);
		if (FSByteStringCompare(type, bsHighlightType) == 0)
		{
			FPD_Annot pdfAnnot = FRAnnotGetPDFAnnot(annot);
			FPD_Object obj = FPDAnnotGetAnnotDict(pdfAnnot);

			FS_ByteString bsContent = FSByteStringNew();
			FPDDictionaryGetString(obj, "Contents", &bsContent);
			FS_ByteString bsCompareContent = FSByteStringNew3("This is initial text",-1);
			if (FSByteStringCompare(bsContent, bsCompareContent) == 0)
			{
				FS_ByteString bsContent = FSByteStringNew3("This is the new text for the
annotation.", -1);
				FPDDictionarySetAtString(obj, "Contents", bsContent);
				FR_MarkupPanel panel = FRMarkupPanelGetMarkupPanel();
				FRMarkupPanelRefreshAnnot(panel, annot, TRUE);
			}
			FSByteStringDestroy(bsContent);
			FSByteStringDestroy(bsCompareContent);
		}
		FSWideStringDestroy(bsHighlightType);
}

FRDocReloadPage(frDocument, 0, FALSE);
```

Deleting specified type annotations

You can delete specify annotation after you get it. For example, you can retrieve an existing highlight annotation and delete it.

Before you delete an annotation, determine whether the annotation is the match subtype. That is, before delete a highlight annotation, ensure that the annotation is a highlight annotation. You can determine whether an annotation is the correct subtype by invoking the FRAnnotGetType method. This method requires a FR_Annot object and return FS_ByteString object that specifies the annotation's subtype. It's the process with modify annotations.

Last the same is needed to use FRMarkupPanelReloadAnnots to reload annotations.

You can use the Foxit API to delete specified annotations by performing the following tasks:

- Get count of the current page by FRPageViewCountAnnot.
- Get type of the annot by iterate all the annotations in the page.

- Delete annotations.
- Reload annot for comments panel.

Example: Delete annot

```
FR_Document frDocument = FRAppGetActiveDocOfPDDoc();
FR_DocView frDocView = FRDocGetCurrentDocView(frDocument);
FR_PageView frPageView = FRDocViewGetCurrentPageView(frDocView);
FS_INT32 count = FRPageViewCountAnnot(frPageView);
for (int i = 0; i < count; i++)
{
		FR_Annot annot = FRPageViewGetAnnotByIndex(frPageView, i);
		FS_ByteString type = FSByteStringNew();
		FRAnnotGetType(annot, &type);
		FS_ByteString bsHighlightType = FSByteStringNew3("Highlight", -1);
		FS_INT32 bSame = FSByteStringCompare(type, bsHighlightType);
		FSByteStringDestroy(type);
		FSByteStringDestroy(bsHighlightType);
		if (bSame == 0)
		{
			FRPageViewDeleteAnnot(frPageView, annot);

			FR_MarkupPanel panel = FRMarkupPanelGetMarkupPanel();
			FRMarkupPanelReloadAnnots(panel, frDocument);
			FRDocReloadPage(frDocument, 0, FALSE);
			break;
		}
}
```

Working with redaction annotations

The Foxit Plug-in SDK API lets you create redaction annotations and modify the attributes in an existing redaction annotation, it's the same with highlight annotations. It also lets you apply an existing redaction annotation, which permanently removes the redacted material from the PDF document.

A redaction annotation identifies content to be removed from the document. The intent of redaction annotations is to enable the following process:

- Create redaction annotations that identify the content to be removed from the document. The redaction annotation specifies a rectangle that covers the content to be removed and specifies the appearance of the rectangle and associated information.
- Apply redaction annotations, which remove the content in the area specified by a set of redaction annotations. In the removed content's place, some marking appears to indicate

that the area was redacted. Also, the redaction annotations are removed from the PDF document.

Creating a redaction annotation

To create a redaction annotation that identifies the content to be removed from the document and the appearance of the redaction annotation, the process is the same with creating Highlight annotations.

Example: Create a redaction annotation

```
FR_Document frDocument = FRAppGetActiveDocOfPDDoc();
FPD_Document fpdDocument = FRDocGetPDDoc(frDocument);
FR_DocView frDocView = FRDocGetCurrentDocView(frDocument);
FR_PageView frPageView = FRDocViewGetCurrentPageView(frDocView);
FPD_Object fpdObject = FPDDictionaryNew();
FS_ByteString strtype = FSByteStringNew3("Annot", -1);
FPD_Object fpdfstringAnnot = FPDStringNew(strtype, 0);
FPDDictionaryAddValue(fpdObject, "Type", fpdfstringAnnot);
FS_ByteString strSubtype = FSByteStringNew3("Redact", -1);
FPD_Object fpdfstringSubtype = FPDStringNew(strSubtype, 0);
FPDDictionaryAddValue(fpdObject, "Subtype", fpdfstringSubtype);
FPDDictionaryAddValue(fpdObject, "Subj", FPDStringNew(FSByteStringNew3("Redact", -1), 0));
FS_FloatRect rectFirst;
rectFirst.left = 100;
rectFirst.right = 300;
rectFirst.top = 500;
rectFirst.bottom = 300;
FPD_Object quadColor = FPDArrayNew();
FS_COLORREF color = RGB(220,20,60);
FPDArrayAddNumber(quadColor, (FS_FLOAT)GetRValue(color) / 255.0f); FPDArrayAddNumber(quadColor,
(FS_FLOAT)GetGValue(color) / 255.0f); FPDArrayAddNumber(quadColor, (FS_FLOAT)GetBValue(color) / 255.0f);
FPDDictionaryAddValue(fpdObject, "C", quadColor);
FPDDictionaryAddValue(fpdObject, "IC", quadColor);

FPDDictionarySetAtRect(fpdObject, "Rect", rectFirst);
FR_Annot frAnnot = FRPageViewAddAnnot(frPageView, fpdObject, 0);
FRDocReloadPage(frDocument, 0, FALSE);
```

Applying redaction annotations

To create a redaction annotation that identifies the content to be removed from the document and the appearance of the redaction annotation, the process is same with create Highlight annot.

To apply previously created redaction annotations, perform the following tasks:

1. After create redaction annotation, you can get FR_Annotobject through FRPageViewGetAnnotByIndex.
2. Apply redaction of the document by FRRedactionApply.

Example: Apply a redaction annotation

```
FR_Document frDocument = FRAppGetActiveDocOfPDDoc();
FPD_Document fpdDocument = FRDocGetPDDoc(frDocument);
FR_DocView frDocView = FRDocGetCurrentDocView(frDocument);
FR_PageView frPageView = FRDocViewGetCurrentPageView(frDocView);
FS_INT32 count = FRPageViewCountAnnot(frPageView);

FS_PtrArray annotArr = FSPtrArrayNew();
for (int i = 0; i < count; i++)
{
		FR_Annot annot = FRPageViewGetAnnotByIndex(frPageView, i);
		FS_ByteString type = FSByteStringNew();
		FRAnnotGetType(annot, &type);
		FS_ByteString bsRedactType = FSByteStringNew3("Redact", -1);
		if (FSByteStringCompare(type, bsRedactType) == 0)
		{
			FSPtrArrayAdd(annotArr, annot);
		}
		FSByteStringDestroy(type);
		FSByteStringDestroy(bsRedactType);
}
FRRedactionApply(frDocument, annotArr, TRUE);
FSPtrArrayDestroy(annotArr);
```

Working with Bookmarks

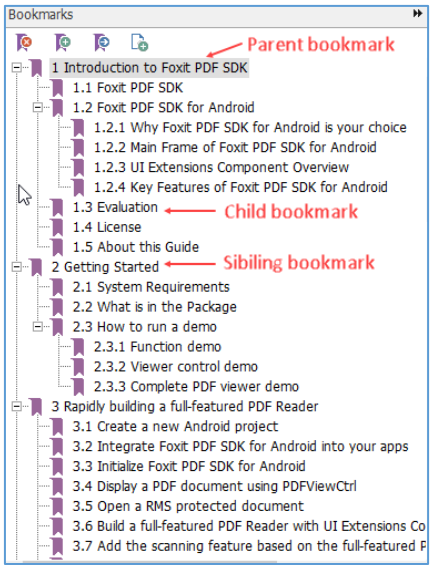
About bookmarks

Bookmarks are represented by a **FPD_Bookmark** object. All bookmarks have the following attributes:

- A title that appears in Foxit PDF Reader/Editor.
- An action that specifies what happens when a user clicks on the bookmark. The typical action for a bookmark is to move to another location in the current document, although other actions can be specified.

Every document has a root bookmark. The root bookmark does not represent a physical bookmark that appears in Foxit PDF Reader/Editor, but is the root from which all bookmarks in the tree are descended. Bookmarks are organized in a tree structure in which each bookmark has zero or more children that appear indented, and zero or more siblings that appear at the same indentation level. All bookmarks except the bookmark at the top level of the hierarchy have a parent, the bookmark under which it is indented. A bookmark is open if its children are visible on screen, and closed if they are not.

The following image shows how bookmarks appear in Foxit PDF Reader/Editor.



The Foxit Plug-in SDK API contains methods that operate on bookmarks. Using these methods, you can perform the following tasks:

- Create new bookmarks
- Get and set various attributes of a bookmark (such as its title or action or whether it is open)
- Search for a bookmark

Creating bookmarks

Before you can create a bookmark, you must create a FPD_Document object that represents the PDF document to which the bookmark is added.

To create bookmarks for a PDF document, perform the following tasks:

1. Get the root of the PDF document's bookmark tree by invoking the FPDDocGetRoot method. This method requires a FPD_Document object and returns a FPD_Object object that represents the node information of the root bookmark of the document. The document's root bookmark does not appear in Foxit PDF Reader/Editor.
2. After configuring and constructing the FPD_Object object, by calling the FPDBookmarkNew method, you can create an FPD_Bookmark bookmark object to be added to the document. There is a Parent field in the FPD_Object object, which is used to set the bookmark object relationship.

The following code example adds two new bookmarks to a PDF document. After each bookmark is created, the FPDBookmarkIsVaild method is invoked to determine whether the bookmark is valid. The name of the FPD_Document object used in this code example is m_pDestDoc. The name of the document root bookmark object used in this code example is m_pRootBookmark.

Example: Creating bookmarks

```
// Declare a bookmark object.
FPD_Bookmark rootBookmark;
FPD_Bookmark childBookmark;
FPD_Bookmark siblingBookmark;

// Get the root bookmark.
rootBookmark = InitDocRootBookmark(myPDDoc);
if (FPDBookmarkIsVaild(rootBookmark))
{
        // Add a child bookmark to the root bookmark.
        childBookmark = AddChildBookmark(rootBookmark, "ChildBookmark");
        if (FPDBookmarkIsVaild(childBookmark))
        {
```

```
                    // Add a sibling bookmark to the child bookmark.
                    siblingBookmark = AddSiblingBookmark(childBookmark, "SiblingBookmark");
            }
}
```

tips:

- InitDocRootBookmark method see: Getting document root bookmark
- AddChildBookmark method see: Adding child bookmark
- AddSiblingBookmark method see: Adding sibling bookmark

Getting the root bookmark of the document

Every PDF document has a root bookmark. The root bookmark does not represent a physical bookmark, but is the root from which all bookmarks in the tree are descended.

The main steps to get the root bookmark of the document are as follows:

- Get the document root dictionary information through FPDDocGetRoot. This method requires an FPD_Document object as the document object to be operated on.
- Determine whether there is bookmark information (ie "Outlines" dictionary item) in the document root dictionary information, if not, create it through FPDDictionaryNew and related methods.
- Create a document root bookmark object based on the obtained or created root dictionary information and through the FPDBookmarkNew method.

The following code example creates a user-defined function named InitDocRootBookmark shows how to get a PDF document's root bookmark.

Example: Get the root bookmark of the document

```
FPD_Bookmark InitDocRootBookmark()
{
   FPD_Object hRootDic = FPDDocGetRoot(m_pDestDoc);
   if(FALSE == FPDDictionaryKeyExist(hRootDic, "Outlines"))
   {
      FPD_Object hParentDic = FPDDictionaryNew();
      FS_DWORD ParentObjNum = FPDDocAddIndirectObject(m_pDestDoc, hParentDic);
      FPDDictionarySetAtReferenceToDoc(hRootDic, "Outlines", m_pDestDoc, ParentObjNum);

      FPDDictionarySetAtName(hParentDic, "Type", "Outlines");
      FPDDictionarySetAtNumber(hParentDic, "Count", 0);

      return FPDBookmarkNew(hParentDic);
   }
```

```
    else
    {
        return FPDBookmarkNew(hRootDic);
    }
}
```

Tips: The name of the FPD_Document object used in this code example is m_pDestDoc.

Adding child bookmark

Adding a new bookmark is mainly divided into adding a child bookmark and adding a sibling bookmark. Here, the method of adding a child bookmark is mainly explained, and the adding position is the end of the child bookmark.

The main steps to add child bookmarks are as follows:

- Create a new FPD_Object dictionary object for subsequent creation of FPD_Bookmark objects. The FPD_Bookmark object needs to be created based on the dictionary object (See: Adding New bookmark dictionary).
- Bind the parent bookmark dictionary object to the newly created FPD_Object dictionary object and set it as the parent dictionary object (i.e. "Parent" dictionary item).
- Update the last dictionary object of the parent bookmark to the new FPD_Object dictionary object (ie "Last" dictionary item). If there is no child bookmark in the parent bookmark, you need to set the first dictionary object of the parent bookmark to the new FPD_Object dictionary object at the same time (Ie "First" dictionary item), if there is a child bookmark, you need to get the original last item child bookmark, and bind the original last item child bookmark dictionary object with the new FPD_Object dictionary object (ie "Next" dictionary item of the original last child bookmark and "Prev" dictionary item of new bookmark).
- Update the dictionary item of the number of children of the parent bookmark (See: Adding the child count of the parent bookmarks).
- Create a new FPD_Bookmark object that is a new bookmark object based on the newly created FPD_Object dictionary object.

The following code example creates a user-defined function named AddChildBookmark. This method requires the following parameters:

- The FPD_Bookmark object represents the parent bookmark to which the child bookmark is to be added. If NULL is passed in, it means the root bookmark of the PDF document.
- Specify the character pointer of the bookmark title.

Example: Add child bookmark

```
FPD_Bookmark AddChildBookmark(FPD_Bookmark hParent, FS_LPCSTR csTitle)
{
   FPD_Bookmark pChildBookmark = FPDBookmarkNew(NULL);
   if(m_pDoc == nullptr ||m_pDestDoc == nullptr)
   {
      return pChildBookmark;
   }

   if(hParent == nullptr)
   {
      hParent = m_pRootBookmark;
   }

   FPD_Object hParentDic = FPDBookmarkGetDictionary(hParent);

   FPD_Object hNewDic = AddNewBookmarkObject(hParentDic, csTitle);
   FS_DWORD NewBMObjnum = FPDDocAddIndirectObject(m_pDestDoc, hNewDic);

   FS_DWORD ParentObjNum = FPDDocAddIndirectObject(m_pDestDoc, hParentDic);
   FPDDictionarySetAtReferenceToDoc(hNewDic, "Parent", m_pDestDoc, ParentObjNum);

   FPD_Object hFirstChildDic = FPDDictionaryGetDict(hParentDic, "First");
   if(hFirstChildDic == nullptr)
   {
      FPDDictionarySetAtReferenceToDoc(hParentDic, "First", m_pDestDoc, NewBMObjnum);
   }

   FPD_Object hLastChildDic = FPDDictionaryGetDict(hParentDic, "Last");
   if(hLastChildDic != nullptr)
   {
      FS_DWORD LastObjNum = FPDDocAddIndirectObject(m_pDestDoc, hLastChildDic);
      FPDDictionarySetAtReferenceToDoc(hLastChildDic, "Next", m_pDestDoc, NewBMObjnum);

      FPDDictionarySetAtReferenceToDoc(hNewDic, "Prev", m_pDestDoc, LastObjNum);
   }

   FPDDictionarySetAtReferenceToDoc(hParentDic, "Last", m_pDestDoc, NewBMObjnum);

   AddParentBookmarkCount(hParentDic);
   pChildBookmark = FPDBookmarkNew(hNewDic);
   return pChildBookmark;
}
```

Tip: After the above creation steps are executed, you need to call the FPDDocSave method to save before it can take effect.

Adding sibling bookmark

Adding a new bookmark is mainly divided into adding a child bookmark and adding a sibling bookmark. Here, the method of adding a sibling bookmark is mainly explained, and the added position is behind the bookmark of the same level passed in the parameter.

The main steps to add sibling bookmarks are as follows:

- Get the dictionary information of the incoming bookmark through the FPDBookmarkGetDictionary method.
- Use the FPDDictionaryGetDict method to obtain the dictionary information of the parent bookmark (ie the "Parent" dictionary item) and the dictionary information of the next bookmark at the same level (ie the "Next" dictionary item) according to the dictionary information of the incoming bookmark.
- Create a new FPD_Object dictionary object for subsequent creation of FPD_Bookmark objects. The FPD_Bookmark object needs to be created based on the dictionary object (see: Adding New bookmark dictionary).
- Bind the obtained parent bookmark dictionary object to the newly created FPD_Object dictionary object, and set it as the parent dictionary object (ie "Parent" dictionary item).
- Bind the incoming bookmark dictionary object to the newly created FPD_Object dictionary object, and set it to the previous dictionary object at the same level (ie "Prev" dictionary item).
- Update the next bookmark dictionary information of the same level in the incoming bookmark dictionary object to the newly created FPD_Object dictionary object (ie "Next" dictionary item).
- If the dictionary object of the next bookmark of the same level originally exists, it will be bound to the newly created FPD_Object dictionary object and set to the next dictionary object of the same level (ie "Next" dictionary item). At the same time, the information of the previous bookmark dictionary at the same level in the original next bookmark dictionary object at the same level is updated to the newly created FPD_Object dictionary object (ie "Prev" dictionary item).
- The dictionary item that updates the number of children of the parent bookmark (See: Adding the child count of the parent bookmark).
- Create a new FPD_Bookmark object that is a new bookmark object based on the newly created FPD_Object dictionary object

The following code example creates a user-defined function named AddSiblingBookmark. This method requires the following parameters:

- The FPD_Bookmark object represents the bookmark to which the same-level bookmark is to be added. If it is passed in, it is invalid if NULL is passed in.
- Specify the character pointer of the bookmark title.

Example: Add sibling bookmark

```
FPD_Bookmark AddSiblingBookmark(FPD_Bookmark hPreBookmark, FS_LPCSTR csTitle)
{
   FPD_Bookmark pSiblingBookmark = FPDBookmarkNew(NULL);
   if(hPreBookmark == nullptr || m_pDoc == nullptr ||m_pDestDoc == nullptr)
   {
      return pSiblingBookmark;
   }

   FPD_Object hPreDic = FPDBookmarkGetDictionary(hPreBookmark);
   FPD_Object hNextDic = FPDDictionaryGetDict(hPreDic, "Next");
   FPD_Object hParentDic = FPDDictionaryGetDict(hPreDic, "Parent");


   FPD_Object hNewDic = AddNewBookmarkObject(hParentDic, csTitle);
   FS_DWORD NewBMObjnum = FPDDocAddIndirectObject(m_pDestDoc, hNewDic);

   FS_DWORD ParentObjNum = FPDDocAddIndirectObject(m_pDestDoc, hParentDic);
   FPDDictionarySetAtReferenceToDoc(hNewDic, "Parent", m_pDestDoc, ParentObjNum);

   FS_DWORD PreObjNum = FPDDocAddIndirectObject(m_pDestDoc, hPreDic);
   FPDDictionarySetAtReferenceToDoc(hPreDic, "Next", m_pDestDoc, NewBMObjnum);
   FPDDictionarySetAtReferenceToDoc(hNewDic, "Prev", m_pDestDoc, PreObjNum);

   if(hNextDic)
   {

      FS_DWORD NextObjNum = FPDDocAddIndirectObject(m_pDestDoc, hNextDic);
      FPDDictionarySetAtReferenceToDoc(hNextDic, "Prev", m_pDestDoc, NewBMObjnum);
      FPDDictionarySetAtReferenceToDoc(hNewDic, "Next", m_pDestDoc, NextObjNum);
   }
   else
   {
      FPDDictionarySetAtReferenceToDoc(hParentDic, "Last", m_pDestDoc, NewBMObjnum);
   }

   AddParentBookmarkCount(hParentDic);
   pSiblingBookmark = FPDBookmarkNew(hNewDic);
   return pSiblingBookmark;
}
```

Tip: After the above creation steps are executed, you need to call the FPDDocSave method to save before it can take effect.

Adding New bookmark dictionary

A new bookmark can be created through FPDBookmarkNew. This method requires an FPD_Object parameter, which represents the dictionary information object of this bookmark. Therefore, before creating a new bookmark, we need to construct the dictionary information object of this new bookmark.

The main steps to create a new bookmark dictionary information are as follows:

- Create a FPD_Object parameter, which represents the PDF target object of the bookmark (see: Creating PDF Destination object).
- Create another FPD_Object parameter to represent the action object of the bookmark (see: Defining bookmark actions)
- Finally, create a FPD_Object parameter, which represents the dictionary object of the bookmark, and bind the created bookmark action object.

The following code example creates a user-defined function named AddNewBookmarkObject. This method requires the following parameters:

- An FPD_Object object, representing the parent bookmark dictionary object of the created bookmark dictionary object.
- Specify the character pointer of the bookmark title.

Example: Add a new bookmark dictionary

```
FPD_Object AddNewBookmarkObject(FPD_Object hParentDic, FS_LPCSTR csTitle)
{
   FS_INT32 nPage = -1;
   FR_PAGESTATE state;
   FRDocGetTopPageState(m_pDoc, &nPage, &state);

   FPD_Object hNewDicDest = FPDArrayNew();
   FS_DWORD PageObjNum = FPDDocAddIndirectObject(m_pDestDoc, FPDDocGetPage(m_pDestDoc, nPage));
   FPDArrayAddReferenceToDoc(hNewDicDest, m_pDestDoc, PageObjNum);
   switch (state.nFitType)
   {
      case FPD_ZOOM_XYZ:
         {
            FPDArrayAddName(hNewDicDest, "XYZ");
            FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 0));
            FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 1));
            FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 2));
         }
         break;
      case FPD_ZOOM_FITPAGE:
```

```
        {
            FPDArrayAddName(hNewDicDest, "Fit");
        }
        break;
    case FPD_ZOOM_FITHORZ:
        {
            FPDArrayAddName(hNewDicDest, "FitH");
            FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 0));
        }
        break;
    case FPD_ZOOM_FITVERT:
        {
            FPDArrayAddName(hNewDicDest, "FitV");
            FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 0));
        }
        break;
    case FPD_ZOOM_FITRECT:
        {
            FPDArrayAddName(hNewDicDest, "FitR");
            FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 0));
            FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 1));
            FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 2));
            FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 3));
        }
        break;
    case FPD_ZOOM_FITBHORZ:
        {
            FPDArrayAddName(hNewDicDest, "FitBH");
            FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 0));
        }
        break;
    default:
        break;
}

FPD_Object hNewDicAction = FPDDictionaryNew();
FPDDictionarySetAtName(hNewDicAction, "Type", "Action");
FPDDictionarySetAtName(hNewDicAction, "S", "GoTo");
FS_DWORD NewDestObjNum = FPDDocAddIndirectObject(m_pDestDoc, hNewDicDest);
FPDDictionarySetAtReferenceToDoc(hNewDicAction, "D", m_pDestDoc, NewDestObjNum);

FPD_Object hNewDic = FPDDictionaryNew();

FS_ByteString bsTitle = FSByteStringNew();
FSByteStringFill(bsTitle, csTitle);
FPDDictionarySetAtString(hNewDic,"Title", bsTitle);
FSByteStringDestroy(bsTitle);
FS_DWORD NewActionObjNum = FPDDocAddIndirectObject(m_pDestDoc, hNewDicAction);
FPDDictionarySetAtReferenceToDoc(hNewDicAction, "A", m_pDestDoc, NewActionObjNum);

FS_DWORD ParentObjNum = FPDDocAddIndirectObject(m_pDestDoc, hParentDic);
```

```
    FPDDictionarySetAtReferenceToDoc(hNewDicAction, "Parent", m_pDestDoc, ParentObjNum);

    return hNewDic;
}
```

Tips: After creating a new bookmark dictionary, you can use the FPDBookmarkNew method to create a new FPD_Bookmark object.

Adding the child count of the parent bookmark

After creating a new bookmark dictionary object and configuring the relevant dictionary item information correctly, you need to update the number of child items of the parent dictionary object (ie "Count" dictionary item)

The following code example creates a user-defined function named AddParentBookmarkCount. This method requires the following parameters:

- The FPD_Object object represents the parent bookmark dictionary object that needs to update the number of children.

Example: Add parent bookmark child count

```
void AddParentBookmarkCount(FPD_Object hParentDic)
{
    int count = FPDDictionaryGetInteger(hParentDic, "Count");
    if(FPDDictionaryKeyExist(hParentDic, "Title"))
    {
        if (count <= 0)
        {
            count -= 1;
        }
        else
        {
            count += 1;
        }
    }
    else
    {
        if (count < 0)
        {
            count -= 1;
        }
        else
        {
            count += 1;
        }
    }
```

```
    FPDDictionarySetAtNumber(hParentDic, "Count", count);
}
```

Tips: The "+" sign of the number of sub-items indicates that the information of the sub-item is expanding, and the "-" sign indicates that the information of the sub-item has been retracted, and its absolute value is the number of visible sub-items when the outline item has been opened.

Defining bookmark actions

After you create a new bookmark, you must define an action that occurs when a user clicks on the bookmark. Otherwise, nothing occurs when a user clicks on the bookmark.

To create an action for a bookmark, you must create a FPD_Object object that represents the action that occurs when a user clicks on a bookmark. Once you create a FPD_Object object, you can assign it to a bookmark. (See "Assigning an action to a bookmark")

As specified earlier in this chapter, a typical bookmark action is to move to another location in the current document. To illustrate how to create a bookmark action, this section defines a bookmark action that displays a specific page in a PDF document when a user clicks the bookmark.

To define a bookmark action that generates a specific view of a PDF document, you create a FPD_Object object by invoking the FPDDictionaryNew method.

The following code example Create a FPD_Object object that represents the action.

- The name of the FPD_Document object used in this code example is **m_pDestDoc**.
- The name of the PDF destination to which the bookmark jumps used in this code example is hNewDicDest. (See "Create PDF Destination Object").

Example: Create a FPD_Object as bookmark action

```
FPD_Object hNewDicAction = FPDDictionaryNew();
FPDDictionarySetAtName(hNewDicAction, "Type", "Action");
FPDDictionarySetAtName(hNewDicAction, "S", "GoTo");
FS_DWORD NewDestObjNum = FPDDocAddIndirectObject(m_pDestDoc, hNewDicDest);
FPDDictionarySetAtReferenceToDoc(hNewDicAction, "D", m_pDestDoc, NewDestObjNum);
```

Creating PDF Destination object

You must create a FPD_Object object represents a specific view destination in the PDF document in order to create a FPD_Object as bookmark action.

Create an FPD_Object object to represent a specific view target in the PDF document. You need to pass the document page number and document destination location information. For example, you

can get the current document display page number and view destination information by calling the FRDocGetTopPageState method.

The following code example Create a FPD_Object object that represents a specific view destination in the PDF document.

- The name of the FPD_Document object used in this code example is m_pDestDoc.

Example: Create a FPD_Object as a specific view destination in the PDF document

```
FS_INT32 nPage = -1;
FR_PAGESTATE state;
FRDocGetTopPageState(m_pDoc, &nPage, &state);

FPD_Object hNewDicDest = FPDArrayNew();
FS_DWORD PageObjNum = FPDDocAddIndirectObject(m_pDestDoc, FPDDocGetPage(m_pDestDoc, nPage));
FPDArrayAddReferenceToDoc(hNewDicDest, m_pDestDoc, PageObjNum);
switch (state.nFitType)
{
   case FPD_ZOOM_XYZ:
                              {
         FPDArrayAddName(hNewDicDest, "XYZ");
         FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 0));
         FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 1));
         FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 2));
      }
      break;
   case FPD_ZOOM_FITPAGE:
      {
         FPDArrayAddName(hNewDicDest, "Fit");
      }
      break;
   case FPD_ZOOM_FITHORZ:
      {
         FPDArrayAddName(hNewDicDest, "FitH");
         FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 0));
      }
      break;
   case FPD_ZOOM_FITVERT:
     {
        FPDArrayAddName(hNewDicDest, "FitV");
        FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 0));
      }
      break;
   case FPD_ZOOM_FITRECT:
      {
         FPDArrayAddName(hNewDicDest, "FitR");
         FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 0));
         FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 1));
         FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 2));
```

```
        FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 3));
    }
    break;
  case FPD_ZOOM_FITBHORZ:
    {
        FPDArrayAddName(hNewDicDest, "FitBH");
        FPDArrayAddNumber(hNewDicDest, FSFloatArrayGetAt(state.dest, 0));
    }
    break;
  default:
    break;
}
```

The following table specifies the fitting type value that you pass to FPD_Object object that represents a specific view destination in the PDF document.

Value	Description
XYZ	Destination specified as upper-left corner point and a zoom factor.
Fit	Fits the page into the window, corresponding to the viewer's Fit Page menu item.
FitH	Fits the width of the page into the window, corresponding to the viewer's Fit Width menu item.
FitV	Fits the height of the page into a window.
FitR	Fits the rectangle specified by its upper-left and lower-right corner points into the window
FitB	Fits the rectangle containing all visible elements on the page (known as the bounding box) into the window (corresponds to the viewer's Fit Visible menu item).
FitBH	Fits the width of the bounding box into the window.
FitBV	Fits the height of the bounding box into the window.

Assigning an action to a bookmark

After create a FPD_Object as a specific view destination in the PDF document, you can define the bookmark actions object and assign it to a specific bookmark.

The following code example binds the bookmark actions object to a bookmark object.

- The name of the FPD_Document object used in this code example is m_pDestDoc.
- The name of the bookmark actions object used in this code example is hNewDicAction. (See "Defining bookmark actions")

Example: Assigning an action to a bookmark

```
FPD_Object hNewDic = FPDDictionaryNew();
FS_DWORD NewActionObjNum = FPDDocAddIndirectObject(m_pDestDoc, hNewDicAction);
FPDDictionarySetAtReferenceToDoc(hNewDicAction, "A", m_pDestDoc, NewActionObjNum);
FPD_Bookmark pBookmark = FPDBookmarkNew(hNewDic);
```

Retrieving bookmarks

You can retrieve the root bookmark, retrieve a specific bookmark, or retrieve all bookmarks that are located within a PDF document.

Retrieving the root bookmark

Every PDF document has a root bookmark. The root bookmark does not represent a physical bookmark, but is the root from which all bookmarks in the tree are descended. (See "Get doc root bookmark")

The following code example demonstrates how to get the first child bookmark of the root bookmark of a PDF document by calling the FPDBookmarkGetFirstChild method. This method requires the following parameters:

- An FPD_Document object, which represents the PDF document from which the root bookmark is retrieved.
- An FPD_Bookmark object, which represents the parent bookmark. If NULL is passed in, it represents the root bookmark of the PDF document.
- The obtained FPD_Bookmark object, this parameter is an output parameter, and the obtained child bookmark object is stored here.

When calling the FPDBookmarkGetFirstChild method to get the first child bookmark, if there is no bookmark, it will return FALSE, and the returned third parameter child bookmark object is also invalid.

Example: Retrieving the root bookmark

```
FPD_Bookmark pChild = FPDBookmarkNew(NULL);
FS_BOOL bGet = FPDBookmarkGetFirstChild(m_pDestDoc, NULL, &pChild);
```

Retrieving a specific bookmark

You can retrieve a specific bookmark by specifying its title. The following code example uses the incoming parent bookmark as the starting point of the search, and then retrieves the specific bookmark by traversing its children and calling the FPDBookmarkGetTitle method to obtain the first bookmark whose title matches the specified title. This method requires the following parameters:

- A certain level of bookmarks in the bookmark tree as the starting point.
- Specify the character pointer of the bookmark title.

Example: Retrieving a specific bookmark

```
FPD_Bookmark FindBookmark(FPD_Bookmark hParent, FS_LPCSTR csTitle)
{
   FS_WideString wsTitle = FSWideStringNew();
   FPD_Bookmark pChild = FPDBookmarkNew(NULL);
   FS_BOOL bGet = FPDBookmarkGetFirstChild(m_pDestDoc, hParent, &pChild);
   if(bGet)
   {
      FPDBookmarkGetTitle(pChild, &wsTitle);
      QString strTitle = QString::fromStdWString(FSWideStringCastToLPCWSTR(wsTitle));
      if(strTitle.compare(QString::fromStdString(csTitle)) == 0)
      {
         return pChild;
      }
      else
      {
         FPD_Bookmark pBookmark = FindBookmark(pChild, csTitle);
         if(FPDBookmarkIsVaild(pBookmark))
         {
            return pBookmark;
         }
      }

      while(1)
      {
         FPD_Bookmark pNextChild = FPDBookmarkNew(NULL);
         bGet = FPDBookmarkGetNextSibling(m_pDestDoc, pChild, &pNextChild);
         if(bGet)
         {
            FPDBookmarkGetTitle(pNextChild, &wsTitle);
            strTitle = QString::fromStdWString(FSWideStringCastToLPCWSTR(wsTitle));
            if(strTitle.compare(QString::fromStdString(csTitle)) == 0)
            {
               return pNextChild;
            }
            else
            {
               FPD_Bookmark pBookmark = FindBookmark(pNextChild, csTitle);
               if(FPDBookmarkIsVaild(pBookmark))
               {
                  return pBookmark;
               }
            }

            pChild = pNextChild;
         }
```

```
        else
        {
            break;
        }
    }
  }

  FSWideStringDestroy(wsTitle);
  return pChild;
}
```

Tips: The name of the FPD_Document object used in this code example is m_pDestDoc.

Retrieving all bookmarks

You can use the Foxit Plug-in SDK API to retrieve all bookmarks located within a PDF document. For example, you can retrieve the title of every bookmark that is located within a PDF document.

The following code example creates a recursive user-defined function named VisitAllBookmarks. First it invokes the PDBookmarkIsValid method to ensure that the bookmark that is passed is valid (When the input parameter is NULL, it represents the root bookmark, and the root bookmark is always valid without judgment.)

Second, this user-defined function retrieves the title of the bookmark by invoking the FPDBookmarkGetTitle method. This method requires the following arguments:

- A PDBookmark object that contains the title to retrieve.
- A FS_WideString output object representing the title of the bookmark.

The FPDBookmarkGetFirstChild method is invoked to determine whether there are child bookmarks under the current bookmark. If there are child bookmarks, A recursive call is made to VisitAllBookmarks (that is, the user-defined method is invoking itself) until there are no more children bookmarks. Then the FPDBookmarkGetNextSibling method is invoked to get a sibling bookmark and the process continues until there are no more bookmarks within the PDF document.

Example: Retrieving all bookmarks

```
FPD_Bookmark VisitAllBookmarks(FPD_Bookmark hParent)
{
  if(hParent != NULL)
  {
    if(FALSE == FPDBookmarkIsVaild(hParent))
    {
      return;
    }
  }
```

```
    FS_WideString wsTitle = FSWideStringNew();
    FPD_Bookmark pChild = FPDBookmarkNew(NULL);
    FS_BOOL bGet = FPDBookmarkGetFirstChild(m_pDestDoc, hParent, &pChild);
    if(bGet)
    {

        FPDBookmarkGetTitle(hBookmark, &wsTitle);
        VisitAllBookmarks(pChild);

        while(1)
        {
            FPD_Bookmark pNextChild = FPDBookmarkNew(NULL);
            bGet = FPDBookmarkGetNextSibling(m_pDestDoc, pChild, &pNextChild);
            if(bGet)
            {
                pChild = pNextChild;
                FPDBookmarkGetTitle(pNextChild, &wsTitle);
                ShowAllChildBookmark(pNextChild);
            }
            else
            {
                break;
            }
        }
    }
                FSWideStringDestroy(wsTitle);
}
```

Tips: The name of the FPD_Document object used in this code example is m_pDestDoc.

Deleting bookmarks

You can delete bookmarks from a PDF document object, which represents the PDF document whose bookmarks are to be deleted.

Deleting the bookmark and its children

When deleting a specific bookmark, you need to update the dictionary objects of its parent bookmark and the adjacent bookmarks at the same level at the same time, and remove the bookmark dictionary object to be deleted from the PDF document, so if the deleted bookmark has a child item Need a recursive way to delete its children one by one.

The following code example creates a recursive user-defined function named DeleteBookmarkAndChild.

First, it calls the PDBookmarkIsValid method to ensure that the bookmark passed is valid.

Then call the FPDBookmarkGetFirstChild method to determine whether there is a child bookmark under the current bookmark.

- If there is no sub-bookmark, you can delete this bookmark (See: Deleting a bookmark).
- Otherwise, make a recursive call to DeleteeBookmarkAndChild (that is, the user-defined method is calling itself) until there are no more child bookmarks. Then call the FPDBookmarkGetNextSibling method to get the bookmark at the same level, and then continue the process until there are no other bookmarks in the PDF document.

Example: Delete bookmark and its children

```
void DeleteBookmarkAndChild(FPD_Bookmark hParent)
{
   if(FALSE == FPDBookmarkIsVaild(hParent))
   {
      return;
   }

   FPD_Bookmark pChild = FPDBookmarkNew(NULL);
   FS_BOOL bGet = FPDBookmarkGetFirstChild(m_pDestDoc, hParent, &pChild);
   if(bGet)
   {
      while(1)
      {
         FPD_Bookmark pNextChild = FPDBookmarkNew(NULL);
         bGet = FPDBookmarkGetNextSibling(m_pDestDoc, pChild, &pNextChild);
         DeleteBookmarkAndChild(pChild);

         if(bGet)
         {
            pChild = pNextChild;
         }
         else
         {
            break;
         }
      }
   }

   DeleteBookmark(hParent);
}
```

Tips: The name of the FPD_Document object used in this code example is m_pDestDoc.

Deleting a bookmark

When deleting a specific bookmark, you need to update the dictionary objects of its parent bookmark and the adjacent bookmarks at the same level at the same time, and remove the

bookmark dictionary object to be deleted from the PDF document. Therefore, if the deleted bookmark does not have a child item It can be deleted through Foxit Plug-in SDK API.

The following code example creates a user-defined function named DeleteBookmark.

- First, it calls the PDBookmarkIsValid method to ensure that the bookmark passed is valid.
- Second, get its own dictionary information object through the FPDBookmarkGetDictionary method.
- Then according to the dictionary items "Prev", "Next" and "Parent" to obtain the previous bookmark dictionary information object at the same level, the next bookmark dictionary information object at the same level and the parent bookmark dictionary object.
- Remove the bookmark dictionary object to be deleted from the PDF document through the FPDDocReleaseIndirectObject and FPDDocDeleteIndirectObject methods.
- If there is no previous bookmark dictionary information object at the same level and the next bookmark dictionary information object at the same level, it means that the parent bookmark has only one child item of the bookmark, and then use the FPDDictionaryRemoveAt method to change the "First" "Last" "Count" dictionary item to delete.
- If there is no previous bookmark dictionary information object at the same level, but there is a next bookmark dictionary information object at the same level, indicating that the bookmark to be deleted is the first child item of its parent bookmark, then the next bookmark dictionary information at the same level is checked through the FPDDictionaryRemoveAt method The "Prev" dictionary item of the object is deleted, and the dictionary information object is updated to the first child item of the parent bookmark dictionary object (ie, the "First" dictionary item) through the FPDDictionarySetAtReferenceToDoc method.
- If there is the previous bookmark dictionary information object at the same level, but there is no next bookmark dictionary information object at the same level, indicating that the bookmark to be deleted is the last child of its parent bookmark, use the FPDDictionaryRemoveAt method to compare the previous bookmark dictionary information object at the same level Delete the "Next" dictionary item of the "Next" dictionary item, and use the FPDDictionarySetAtReferenceToDoc method to update the dictionary information object to the last child item of the parent bookmark dictionary object (ie, the "Last" dictionary item).

- If there are both the previous bookmark dictionary information object at the same level and the next bookmark dictionary information object at the same level, indicating that the bookmark to be deleted is an intermediate child of its parent bookmark, the FPDDictionarySetAtReferenceToDoc method is used to compare the previous bookmark dictionary information object at the same level. The "Next" dictionary item is re-bound to the next same-level bookmark dictionary object, and at the same time, the "Prev" dictionary item of the next same-level bookmark dictionary information object is re-bound to the previous same-level bookmark dictionary object.
- Finally update the number of child items of the parent bookmark dictionary object (ie "Count" dictionary item). If the deleted bookmark is the only child item of the parent bookmark and the dictionary item has been removed, you do not need to operate this step (See: Decreasing the count of parent bookmark).

Example: Delete a bookmark

```
void DeleteBookmark(FPD_Bookmark hDeleteBookmark)
{
   if(FALSE == FPDBookmarkIsVaild(hDeleteBookmark))
   {
      return;
   }

   FPD_Object hDeleteBookmarkDic = FPDBookmarkGetDictionary(hDeleteBookmark);
   FPD_Object hPreBookmarkDic = FPDDictionaryGetDict(hDeleteBookmarkDic, "Prev");
   FPD_Object hNextBookmarkDic = FPDDictionaryGetDict(hDeleteBookmarkDic, "Next");
   FPD_Object hParentDic = FPDDictionaryGetDict(hDeleteBookmarkDic, "Parent");

   FPDDocReleaseIndirectObject(m_pDestDoc, FPDObjectGetobjNum(hDeleteBookmarkDic));
   FPDDocDeleteIndirectObject(m_pDestDoc, FPDObjectGetobjNum(hDeleteBookmarkDic));

   if(hPreBookmarkDic == nullptr  && hNextBookmarkDic == nullptr)
   {
      FPDDictionaryRemoveAt(hParentDic, "First");
      FPDDictionaryRemoveAt(hParentDic, "Last");
      FPDDictionaryRemoveAt(hParentDic, "Count");
   }
   else
   {
      if (hPreBookmarkDic == nullptr)
      {
         FPDDictionaryRemoveAt(hNextBookmarkDic, "Prev");

         FS_DWORD NextObjNum = FPDDocAddIndirectObject(m_pDestDoc, hNextBookmarkDic);
         FPDDictionarySetAtReferenceToDoc(hParentDic, "First", m_pDestDoc, NextObjNum);
      }
      else if(hNextBookmarkDic == nullptr)
```

```
    {
      FPDDictionaryRemoveAt(hPreBookmarkDic, "Next");

      FS_DWORD PreObjNum = FPDDocAddIndirectObject(m_pDestDoc, hPreBookmarkDic);
      FPDDictionarySetAtReferenceToDoc(hParentDic, "Last", m_pDestDoc, PreObjNum);
    }
    else
    {
      FS_DWORD PreObjNum = FPDDocAddIndirectObject(m_pDestDoc, hPreBookmarkDic);
      FPDDictionarySetAtReferenceToDoc(hNextBookmarkDic, "Prev", m_pDestDoc, PreObjNum);

      FS_DWORD NextObjNum = FPDDocAddIndirectObject(m_pDestDoc, hNextBookmarkDic);
      FPDDictionarySetAtReferenceToDoc(hPreBookmarkDic, "Next", m_pDestDoc, NextObjNum);
    }

    DeleteParentBookmarkCount(hParentDic);
  }
}
```

Tips: After the above creation steps are executed, you need to call the FPDDocSave method to save before it can take effect.

Decreasing the count of parent bookmark

After deleting a bookmark dictionary object, you need to update the number of children of the parent dictionary object (ie "Count" dictionary item)

The following code example creates a user-defined function named DeleteParentBookmarkCount. This method requires the following parameters:

The FPD_Object object represents the parent bookmark dictionary object that needs to update the number of children.

```
void DeleteParentBookmarkCount(FPD_Object hParentDic)
{
  int count = FPDDictionaryGetInteger(hParentDic, "Count");
  if(count <= 0)
  {
    count += 1;
  }
  else
  {
    count -= 1;
  }

  FPDDictionarySetAtNumber(hParentDic, "Count", count);
}
```

Tips: The "+" sign of the number of sub-items indicates that the information of the sub-item is expanding, and the "-" sign indicates that the information of the sub-item has been retracted, and its absolute value is the number of visible sub-items when the outline item has been opened.

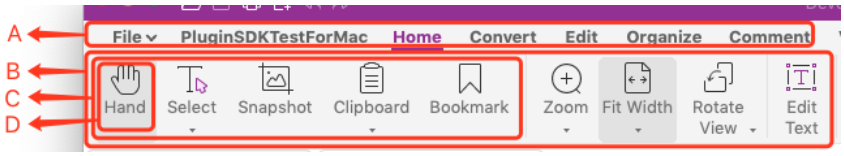
Ribbon Bar and Buttons

This chapter explains how to use Foxit Plug-in SDK API to create new toolbars and toolbar buttons. For example, you can create a new button, attach it to an existing toolbar, and bind the execution function of the click button to a callback function specified by the user.

About Ribbon bar

Foxit PDF Reader/Editor consists of various toolbars that enable a user to invoke specific functionality.

For example, you can click the "Select" button on the **Home** toolbar to select the text in the current opened document. The image below shows the toolbar style in Foxit PDF Editor.



Letter	Description	Match Object Type	Remarks
A	The Ribbon Toolbar	FR_RibbonBar	Ribbon Bar is a Microsoft-office style toolbar. "Ribbon" control was introduced by Microsoft in Office 2007. It's not just a new control - it's a new user interface ideology. Ribbon control replaces traditional toolbars and menus with tabbed groups (Categories). Each group is logically split into Panels and each panel may contain various controls and command buttons. Ribbon control behaves as a "static" (non-floating) control bar and can be docked at the top of frame. the Ribbon Bar hosts many Ribbon Categories. A Ribbon Category is a logical entity. The visual representation of Category is Tab. A Category contains (and the Tab displays) a group of Ribbon Panels. Each Ribbon Panel contains one or more Ribbon Elements.
B	The Ribbon Category	FR_RibbonCategory	Ribbon Category implements the functionality of a logical entity containing a group of panels. The visual representation of Ribbon Category is a Tab.

C	The Ribbon Panel	FR_RibbonPanel	FR_RibbonPanel implements functionality of a single entity that contains a set of ribbon elements. The panel has a special layout logic which allows to display as many elements as possible according to its current size.
D	The Function Button	FR_RibbonElement or FR_CommonControl	FR_RibbonElement is the objects (elements) that can be placed on Ribbon control. For example, ribbon buttons, ribbon check boxes, combo boxes are all ribbon elements. **Note**: FR_RibbonElement is an exclusive Windows platform type. We provide a cross-platform type: FR_CommonControl.

Tips: You can hover over the function button to get a more detailed description of the function button.

Note: Each component of the toolbar is called the title displayed on the Foxit PDF Reader/Editor interface. In addition, there are internal object names for programmatic retrieval of the toolbar, and the displayed title is different from the internal object name. For example, the "Home" toolbar. The display title is the home page, and the internal object name under the Mac is Ribbon_Page_Home. The internal object name can be obtained through Foxit Plug-in SDK API (See "Retrieving Ribbon Category").

Retrieving Ribbon Category

You can use Foxit Plug-in SDK API to retrieve the existing Ribbon Category that appear in Foxit PDF Reader/Editor.

After retrieving the Ribbon Category, you can perform other functions, such as attaching a Ribbon Panel (see "Attaching a Ribbon Panel to Ribbon Category").

You can retrieve specific Ribbon Category components by the following methods.

1. Invoking the **FRRibbonBarGetCategoryByIndex** method, this method requires an index value, the index value is the index position of the Ribbon Category in the Ribbon Bar and returns a Ribbon Category object that corresponds to the Ribbon Category. If the index cannot be found, this method returns NULL.

   Example: Search the Ribbon Category by index

```
FR_RibbonBar hRibbonBar = FRAppGetRibbonBar(NULL);
FR_RibbonCategory hRibbonCategory = FRRibbonBarGetCategoryByIndex(hRibbonBar,  0);
```

Note: The index value must be between 0 and the value obtained by **FRRibbonBarGetCategoryCount**.

2. Invoking the **FRRibbonBarGetCategoryByName** method. This method requires a constant character pointer that specifies the internal object name of a Ribbon Category and returns a Ribbon Category object that corresponds to the Ribbon Category. If the name cannot be found, this method returns NULL.

   Example: Search the Ribbon Category by name

```
const char * ribbonCategory = "Category";
FR_RibbonBar hRibbonBar = FRAppGetRibbonBar(NULL);
FR_RibbonCategory hRibbonCategory =
FRRibbonBarGetCategoryByName(hRibbonBar,  ribbonCategory);
```

   Tips: The internal name of the existing Ribbon Category in Foxit PDF Reader/Editor can be obtained by traversing and calling the FRRibbonCategoryGetName method.

   Example: Traverse to obtain the names of all Ribbon Category internal objects

```
FR_RibbonBar hRibbonBar = FRAppGetRibbonBar(NULL);
FS_INT32 iCategoryCount = FRRibbonBarGetCategoryCount(hRibbonBar);
FS_ByteString ribbonCategoryName = FSByteStringNew();
FR_RibbonCategory hRibbonCategory = NULL;
for(FS_INT32 i=0; i<iCategoryCount; i++)
{
   hRibbonCategory = FRRibbonBarGetCategoryByIndex(hRibbonBar, i);
   if(hRibbonCategory !=  NULL)
        {
            FRRibbonCategoryGetName(hRibbonCategory, &ribbonCategoryName);
        }
}
```

Attaching a Ribbon category to a Ribbon Bar

You can create a new Ribbon Category and attach it to the Ribbon Bar. To create a new Ribbon Category, call the FRRibbonBarAddCategory method and pass the following parameters:

1. Specify the FR_RibbonBar object where the Ribbon Category is added.
2. The character pointer representing the name of the internal object of the Ribbon Category, the object name is used for subsequent retrieval to get the Ribbon Category.
3. Represents the display title of the Ribbon Category, which is displayed on the Ribbon Bar of the program.

```
FR_RibbonBar hRibbonBar = FRAppGetRibbonBar(NULL);
```

```
FS_LPCSTR ribbonCategoryName= "CategoryName" ;
FS_LPCSTR ribbonCategoryTitle= "CategoryTitle" ;
FR_RibbonCategory hRibbonCategory = FRRibbonBarAddCategory(hRibbonBar,  ribbonCategoryName,
ribbonCategoryTitle);
```

Note: The Ribbon Category to be added should not be the same as the existing Ribbon Category title and internal object name.

Retrieving Ribbon Panel

You can use Foxit Plug-in SDK API to retrieve the existing Ribbon Panel that appear in Foxit PDF Reader/Editor.

After retrieving the Ribbon Panel, you can perform other functions, such as attaching a button (see "Attaching a button to a Ribbon Panel").

You can retrieve specific Ribbon Panel components by the following methods.

1. Invoking the **FRRibbonCategoryGetPanelByIndex** method, this method requires an index value, the index position of the Ribbon Panel in the Ribbon Category and returns a Ribbon Panel object that corresponds to the Ribbon Panel. If the index cannot be found, this method returns NULL.

   Example: Search the Ribbon Panel by index

   ```
   FS_LPCSTR ribbonCategory= "Category";
   FR_RibbonBar hRibbonBar = FRAppGetRibbonBar(NULL);
   FR_RibbonCategory hRibbonCategory = FRRibbonBarGetCategoryByName(hRibbonBar, ribbonCategory);
   if(hRibbonCategory != NULL)
   {
           FR_RibbonPanel hRibbonPanel = FRRibbonCategoryGetPanelByIndex(hRibbonCategory, 0);
   }
   ```

   Note: The index value must be between 0 and the value obtained by FRRibbonCategoryGetPanelCount.

2. Invoking the **FRRibbonCategoryGetPanelByName** method. This method requires a constant character pointer that specifies the internal object name of a Ribbon Panel and returns a Ribbon Panel object that corresponds to the Ribbon Panel. If the name cannot be found, this method returns NULL.

   Example: Search the Ribbon Panel by name

   ```
   FS_LPCSTR ribbonCategory= "Category";
   FS_LPCSTR ribbonPanel= "Panel";
   ```

```
FR_RibbonBar hRibbonBar = FRAppGetRibbonBar(NULL);
FR_RibbonCategory hRibbonCategory = FRRibbonBarGetCategoryByName(hRibbonBar,  ribbonCategory);
if(hRibbonCategory != NULL)
{
        FR_RibbonPanel hRibbonPanel =
FRRibbonCategoryGetPanelByName(hRibbonCategory,  ribbonPanel);
}
```

Tips: The internal name of the existing Ribbon Panel in Foxit PDF Reader/Editor can be obtained by traversing and calling the **FRRibbonPanelGetName** method.

Example: Traverse to obtain the names of all Ribbon Panel internal objects from a Ribbon Category

```
FS_LPCSTR ribbonCategory= "Category";
FR_RibbonBar hRibbonBar = FRAppGetRibbonBar(NULL);
FR_RibbonCategory hRibbonCategory = FRRibbonBarGetCategoryByName(hRibbonBar,  ribbonCategory);
if(hRibbonCategory != NULL)
{
        FR_RibbonPanel hRibbonPanel = NULL;
        FS_ByteString ribbonPanelName = FSByteStringNew();
        FS_INT32 iPanelCount = FRRibbonCategoryGetPanelCount(hRibbonCategory);
        for(FS_INT32 i=0; i<iPanelCount; i++)
        {
        hRibbonPanel = FRRibbonCategoryGetPanelByIndex(hRibbonCategory, i);
        if(hRibbonPanel !=  NULL)
                {
        FRRibbonPanelGetName(hRibbonPanel, &ribbonPanelName);
                }
        }
}
```

Attaching a Ribbon Panel to Ribbon Category

You can create a new Ribbon Panel and attach it to the Ribbon Category. To create a new Ribbon Panel, call the **FRRibbonCategoryAddPanel** method and pass the following parameters:

1. Specify the FR_RibbonCategory object where the Ribbon Panel is added.
2. The character pointer representing the name of the internal object of the Ribbon Panel, the object name is used for subsequent retrieval to get the Ribbon Panel.
3. Represents the display title of the Ribbon Panel, which is displayed on the Ribbon Category of the program, now hidden.
4. An **FS_DIBitmap** object that represents the Ribbon Panel's icon. If a Ribbon Panel does not have an icon, the Ribbon Panel appears with a gray background.

```
FS_LPCSTR ribbonCategory= "Category" ;
```

```
FR_RibbonBar hRibbonBar = FRAppGetRibbonBar(NULL);
FR_RibbonCategory hRibbonCategory = FRRibbonBarGetCategoryByName(hRibbonBar,  ribbonCategory);
if(hRibbonCategory != NULL)
{
		FS_LPCSTR ribbonPanelName= "PanelName" ;
		FS_LPCSTR ribbonPanelTitle= "PanelTitle" ;
		FS_DIBitmap bitmap = FSDIBitmapNew();
		FSDIBitmapCreate(bitmap, iWidth, iHeight, FS_DIB_Rgb32, NULL, NULL);
		FSDIBitmapClear(bitmap, 0xffffffff);
		FR_RibbonPanel hRibbonPanel = FRRibbonCategoryAddPanel(hRibbonCategory,  ribbonPanelName,
ribbonPanelTitle, bitmap);
}
```

Note: The Ribbon Panel to be added should not be the same as the existing Ribbon Panel title and internal object name.

Retrieving existing buttons

You can use Foxit Plug-in SDK API to retrieve the existing Function Button that appear in Foxit PDF Reader/Editor.

After retrieving the Function Button, you can perform other functions, such as attaching callback functions (see "Creating button callback functions").

You can retrieve specific Function Button components by the following methods.

1. Invoking the **FRRibbonPanelGetControlByIndex** method, this method requires an index value, the index value is the index position of the Function Button in the Ribbon Panel and returns a Function Button object that corresponds to the Function Button. If the index cannot be found, this method returns NULL.

   Example: Search the Function Button by index

   ```
   FS_LPCSTR ribbonCategory= "Category";
   FS_LPCSTR ribbonPanel= "Panel";
   FR_RibbonBar hRibbonBar = FRAppGetRibbonBar(NULL);
   FR_RibbonCategory hRibbonCategory = FRRibbonBarGetCategoryByName(hRibbonBar,  ribbonCategory);
   if(hRibbonCategory != NULL)
   {
   		FR_RibbonPanel hRibbonPanel =
   FRRibbonCategoryGetPanelByName(hRibbonCategory,  ribbonPanel);
   		if(hRibbonPanel != NULL)
   		{
   		   FR_CommonControl hCommonControl = FRRibbonPanelGetControlByIndex(hRibbonPanel,  0);
   		}
   }
   ```

Note: The index value must be between 0 and the value obtained by FRRibbonPanelGetElementCount.

2. Invoking the **FRRibbonPanelGetControlByName** method. This method requires a constant character pointer that specifies the internal object name of a Function Button and returns a Function Button object that corresponds to the Function Button. If the name cannot be found, this method returns NULL.

Example: Search the Function Button by name

```
FS_LPCSTR ribbonCategory= "Category";
FS_LPCSTR ribbonPanel= "Panel";
FS_LPCSTR controlName = "control";
FR_RibbonBar hRibbonBar = FRAppGetRibbonBar(NULL);
FR_RibbonCategory hRibbonCategory = FRRibbonBarGetCategoryByName(hRibbonBar,  ribbonCategory);
if(hRibbonCategory != NULL)
{
	FR_RibbonPanel hRibbonPanel =
FRRibbonCategoryGetPanelByName(hRibbonCategory,  ribbonPanel);
	if(hRibbonPanel != NULL)
	{
		FR_CommonControl hCommonControl =
FRRibbonPanelGetControlByName(hRibbonPanel,  controlName);
	}
}
```

Tips: The internal name of the existing Function Button in Foxit PDF Reader/Editor can be obtained by traversing and calling the FRCommonControlGetName method.

Example: Traverse to obtain the names of all Funtion Button internal objects from a Ribbon Panel

```
FS_LPCSTR ribbonCategory= "Category";
FS_LPCSTR ribbonPanel= "Panel";
FR_RibbonBar hRibbonBar = FRAppGetRibbonBar(NULL);
FR_RibbonCategory hRibbonCategory = FRRibbonBarGetCategoryByName(hRibbonBar,  ribbonCategory);
if(hRibbonCategory != NULL)
{
	FR_RibbonPanel hRibbonPanel =
FRRibbonCategoryGetPanelByName(hRibbonCategory,  ribbonPanel);
	if(hRibbonPanel != NULL)
	{
		FR_CommonControl hCommonControl = NULL;
		FS_ByteString controlName = FSByteStringNew();
		FS_INT32 iControlCount = FRRibbonCategoryGetPanelCount(hRibbonPanel);
		for(FS_INT32 i=0; i<iPanelCount; i++)
		{
		hCommonControl = FRRibbonBarGetCategoryByIndex(hRibbonPanel, i);
```

```
                    if(hCommonControl !=  NULL)
                            {
                    FRRibbonPanelGetName(hCommonControl, &controlName);
                            }
                    }
            }
}
```

Attaching a button to a Ribbon Panel

You can create a new button and attach it to the Ribbon Panel. To create a new button, call the **FRRibbonPanelAddControl** method and pass the following parameters:

1. Specify the FR_RibbonPanel object where the button is added.
2. **FR_Common_Control_Type** object represents the button type. The 4 button types provided are as follows:

   FR_CommonControl_BUTTON ---- common button type

   FR_CommonControl_DROPDOWNBUTTON ---- drop-down menu button type

   FR_CommonControl_DROPDOWNACTION ---- variable button type with drop-down menu

   FR_CommonControl_CHECKBOX ---- check box button type

3. The character pointer representing the name of the internal object of the button, the object name is used for subsequent retrieval to get the button
4. Represents the display title of the button, which is displayed on the Ribbon Panel of the program.

```
FR_RibbonBar hRibbonBar = FRAppGetRibbonBar(NULL);
FS_LPCSTR ribbonCategory= "Category" ;
FR_RibbonCategory hRibbonCategory = FRRibbonBarGetCategoryByName(hRibbonBar,  ribbonCategory);
if(hRibbonCategory !=  NULL)
{
    const char * ribbonPanel= "Panel" ;
    FR_RibbonPanel hRibbonPanel = FRRibbonCategoryGetPanelByName(hRibbonCategory,  ribbonPanel);
    if(hRibbonPanel !=  NULL)
   {
      const char *controlName= "name" ;
      const char *controlTitle= "title" ;
      FR_CommonControl hCommonControl =
FRRibbonPanelAddControl(hRibbonPanel,  FR_CommonControl_BUTTON, controlName, controlTitle);
   }
}
```

Creating button callback functions

You can create a toolbar button callback function which is invoked by Foxit PDF Reader/Editor. when a user clicks a button. For the purposes of this discussion, a simplistic user-defined function named FRExecuteProc is introduced. This method displays a message box by invoking the printf method. The following code shows the body of the FRExecuteProc function.

```
void MyExecuteProc(void *clientData)
{
	printf("A button was clicked.");
}
```

Note: The data parameter for this and the other callbacks can be used to maintain private data that is used by the callback. Notice that this user-defined function is declared using the FRExecuteProc macros.

After creating the FRExecuteProc method, you can call the FRCommonControlSetExecuteProc method to associate the button with the callback. In other words, when the user clicks the button, Foxit PDF Reader/Editor will call the user-defined function. The FRCommonControlSetExecuteProc method requires the following parameters:

1. An FR_CommonControl object, representing the button associated with the callback.
2. FRExecuteProc object representing the callback function.

The following code example creates a callback function for a button.

Example: Creating a function button callback function

```
void MyExecuteProc(void *clientData)
{
	printf("A button was clicked.");
}

void Createbuttoncallbackfunctions()
{
	FS_LPCSTR ribbonCategory= "Category";
	FS_LPCSTR ribbonPanel= "Panel";
	FS_LPCSTR controlName = "control";
	FR_RibbonBar hRibbonBar = FRAppGetRibbonBar(NULL);
	FR_RibbonCategory hRibbonCategory =
FRRibbonBarGetCategoryByName(hRibbonBar,  ribbonCategory);
	if(hRibbonCategory != NULL)
	{
		FR_RibbonPanel hRibbonPanel =
FRRibbonCategoryGetPanelByName(hRibbonCategory,  ribbonPanel);
```

```
                        if(hRibbonPanel != NULL)
                        {
                                FR_CommonControl hCommonControl =
FRRibbonPanelGetControlByName(hRibbonPanel,  controlName);
                                if(hRibbonPanel != NULL)
                                {
                                        FRCommonControlSetExecuteProc(hCommonControl,
&MyExecuteProc);
                                }
                        }
                }
}
```

Tips: The button also has two other similar callback function binding methods:

1. FRCommonControlSetComputeEnabledProc, this method is used to set the button enable state in different situations, the callback function type is FRComputeEnabledProc.
2. FRCommonControlSetAppearanceSettingProc, this method is triggered when the program display mode changes, the callback function type is FRAppearanceSettingProc, which is currently only available on MacOS.

Registering for Event Notifications

This chapter explains how to register for notification of a specific event. The Foxit Plugin SDK API provides a notification mechanism so that plugins can synchronize their actions with Foxit PDF Reader/Editor. Notifications enable plugins to indicate that they are interested in a specified event (such as the initialization of Foxit PDF Reader/Editor) and to provide a callback function that is invoked by Foxit PDF Reader/Editor each time an event occurs. For example:

1. FRAppRegisterPreferencePageHandler
2. FRAppRegisterSecurityHandler
3. RegisterAppEventHandler
4. RegisterDocHandlerOfPDDoc

Registering event notifications

Register for an event notification when you want your plugin to be notified when a specific event occurs. For example, you can register for a notification when Foxit PDF Reader/Editor has finished initializing. To register for an event notification, you provide a callback function that Foxit PDF Reader/Editor invokes when the event occurs.

You can register for an event notification by performing the following tasks:

- Create a user-defined function that is invoked when the event occurs.
- Set the functions pointer to the Event Callback structure.
- Invoke the Register method to register these functions to Foxit PDF Reader/Editor.

The following code example registers for the event that occurs when Foxit PDF Reader/Editor is activating or changing UI language or downloading or ready to quit.

Example: Registering for an event notification

```
/* Define the callback functions that will be invoked when the app event occurs.
 * Add your own application logic in the callback functions to meet your requirement.
 */
void OnLangUIChange(FS_LPVOID clientData)
{
	// App event occurs: The UI of language changes.
}
```

```
void OnActivateApp(FS_LPVOID clientData, FS_BOOL bActive)
{
	// App event occurs: The app is activated.
}

void WillQuit(FS_LPVOID clientData)
{
	// App event occurs: The app will quit.
}

void OnDownload(FS_LPVOID clientData, FS_LPCSTR lpModuleName)
{
	// App event occurs: Needs to download the updated module.
}
	/* set your own callback functions to EventCallbacks structure*/
	FR_AppEventCallbacksRec appEventCallbacks;
	INIT_CALLBACK_STRUCT(&appEventCallbacks, sizeof(FR_AppEventCallbacksRec));
	appEventCallbacks.lStructSize = sizeof(FR_AppEventCallbacksRec);
	appEventCallbacks.clientData = NULL;
	appEventCallbacks.FRAppOnActivate = &OnActivateApp;
	appEventCallbacks.FRAppOnLangUIChange = &OnLangUIChange;
	appEventCallbacks.FRAppWillQuit = &WillQuit;
	appEventCallbacks.FRAppOnDownload = &OnDownload;
	/* Register the app event handler.*/
	FRAppRegisterAppEventHandler(&appEventCallbacks);
```

Working with Host Function Tables

A Host Function Table (HFT) is a mechanism for managing the Foxit Plug-in SDK methods. It is implemented as a pointer array that stores the addresses of Plug-in SDK methods. The methods are grouped together based on the types of objects they are associated with. Each group of methods has a specific HFT for performing actions on a specific object type. All of these HFTs are managed by the Core HFT manager. The manager indexes the HFTs by category. Foxit PDF Reader/Editor consists of numerous internal HFTs that provide plugins with an efficient way to invoke their methods. Here is a high-level summary of the HFT API method search algorithm.

- The Core HFT Manager uses a category selector to locate the specific HFT.
- The manager then uses the method selector to locate the specific method.

In addition to invoking Plug-in SDK methods, extension HFTs can be created for individual plugins. Extension HFTs allow the methods of a specific plugin to be accessible to all other plugins.

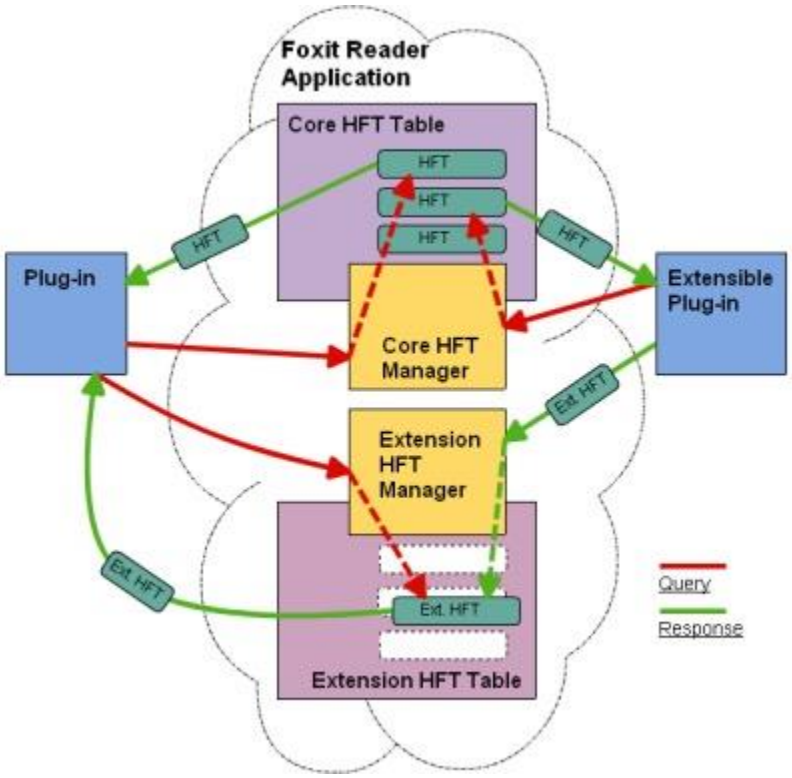
About host function tables

An HFT is a table of function pointers where each HFT contains the following information:

- A name
- A version number
- An array of one or more entries

Each entry represents a single method that a plugin can invoke, and is defined as a linked list of function pointers. Foxit PDF Reader/Editor uses linked lists because some HFT entries may be marked so that they can be replaced by a plugin. Also, it is useful to keep a list of each implementation of a method that was replaced to allow methods to call the implementations they replaced.

The following diagram shows the relationship between Foxit PDF Reader/Editor, other plugins, and HFTs.



Plugins must use the **FSExtensionHFTMgrGetHFT** method to import each HFT they intend to use. A plugin requests an HFT by its name and version number. An HFT is imported during plugin initialization.

When a plugin invokes a method in Foxit PDF Reader/Editor, or another plugin, the function pointer at the appropriate location in the appropriate HFT is dereferenced and executed. Macros in the Foxit Plug-in SDK header files hide this functionality so that plugins contain only what appear to be normal function calls.

Each HFT is serviced by an HFT server. The HFT server is responsible for handling requests to obtain or destroy its HFT. As part of its responsibility to handle requests, an HFT server can choose to support multiple versions of the HFT. These versions generally correspond to versions of Foxit PDF Reader/Editor or the plugin that exposes the HFT.

The ability to provide more than one version of an HFT improves backward-compatibility by allowing existing plugins to continue to work when new versions of Foxit PDF Reader/Editor (or other plugins whose HFTs they use) become available. It is expected that HFT versions typically will differ only in the number, not the order, of methods they contain.

Global Core HFT Manager

User-defined plugins must contain a global Core HFT Manager pointer. If this pointer is not defined, the plugin will fail to compile. This pointer is defined in the PISetupSDK method. PISetupSDK is called by the host application to initialize the plugin.

Example: Get Core HFT Manager

```
/*Core HFT Manager. */
FRCoreHFTMgr *_gpCoreHFTMgr = NULL;

/*
** This routine is called by the host application to set up the Plug-in's SDK-provided functionality.
*/
FS_BOOL PISetupSDK(FS_INT32 handshakeVersion, void *sdkData)
{
   if(handshakeVersion != HANDSHAKE_V0100) return FALSE;
   PISDKData_V0100 *pSDKData = (PISDKData_V0100*) sdkData;

   /* Points to core HFT manage from Foxit PDF Reader/Editor */
   _gpCoreHFTMgr = pSDKData->PISDGetCoreHFT();

   /* Set the plugin's handshake routine, which is called next by the host application */
   pSDKData->PISDSetHandshakeProc(sdkData, &PIHandshake);
   return TRUE;
}
```

Exporting host function tables

You can use Foxit Plug-in SDK API to export HFTs that result in a plugin's methods being available to other plugins. This section will introduce in more detail how to export the host function tables. To make a plugins' set of methods accessible to other plugins, an extension HFT should be created to manage these methods.

You can use the Foxit Plug-in SDK API to export HFTs that result in a plugin's methods being available to other plugins. To export an HFT, perform the following tasks:

1. Create HFT methods.
2. Create HFT method definitions
3. Create a new extension HFT
4. Add the HFT to the host environment.
5. Add the address of the methods to the extension HFT.

Creating HFT methods

The first step in exporting HFTs is to create the methods that will be exported and made available to other plugins. For the purpose of this discussion, assume that the following two methods exist.

Example: Creating HFT methods

```
void Function1()
{

}

void Function2(FS_INT32 iNum)
{

}
```

Creating HFT method definitions

When you invoke a method in an HFT, the methods are accessed through a function pointer. Part of the process of defining a function pointer through which HFT methods are accessed is to create an enumeration that specifies the index of each method that you want to include within an HFT. The following enumeration enables indexing into the HFT.

After you define an enumeration and an HFT object, you can define a function pointer for each method by using the following syntax:

```
typedef return_type (*function_nameSELPROTO)(parameters);
```

The following table describes this syntax.

return_type	The return type of the HFT method
function_name	The name of the HFT method
parameters	The HFT method's parameters with their types

Example: Creating HFT method definitions

```
#undef EXTENSIONMETHODS
#undef BEGINENUM
#undef NumOfSelector
#undef ENDENUM

/*Generate the method selector. */
#define BEGINENUM enum{
#define EXTENSIONMETHODS(returnType, methodName, params) methodName##SEL,
```

```
#define NumOfSelector(name) name##MethodsNum
#define ENDENUM };

BEGINENUM
EXTENSIONMETHODS(void, Function1, ())
EXTENSIONMETHODS(void, Function2, (FS_INT32 iNum))
NumOfSelector(Extension)
ENDENUM

#undef EXTENSIONMETHODS
#undef BEGINENUM
#undef NumOfSelector
#undef ENDENUM

/*Generate the prototypes of the methods.*/
#define BEGINENUM
#define NumOfSelector(name)
#define ENDENUM
#define EXTENSIONMETHODS(returnType, methodName, params) \
typedef returnType (*methodName##PROTO)params;

BEGINENUM
EXTENSIONMETHODS(void, Function1, ())
EXTENSIONMETHODS(void, Function2, (FS_INT32 iNum))
NumOfSelector(Extension)
ENDENUM
```

The indexes are called selectors, hence the SEL at the end of the method names. Function1SEL is the index of Function1 method, Function2SEL is the index of Function2 method.

Also declare a global HFT object that is used in various tasks:

- extern FS_HFT extensionHFT;

For example, to define an HFT method name, you must specify an HFT object. (See "Defining an HFT method name")

Defining an HFT method name

You must specify a name for each method that is used to invoke the HFT method from other plugins. You can define an HFT method name by using the following syntax:

```
#define method_name (*((method_nameSELPROTO)(method_address)))
```

The following table describes the syntax.

method_name	The name of the HFT method that is used to invoke the method from external plugins
method_address	The address of the HFT method that is used to invoke the method from external plugins

For example, to define HFT method name to the Function1 and Function2 methods in the Creating HFT methods step:

```
#define ExtensionHFTFunction1 (*(Function1SELPROTO)(FSExtensionHFTMgrGetEntry(extensionHFT, Function1SEL)))
#define ExtensionHFTFunction2 (*(Function2SELPROTO)(FSExtensionHFTMgrGetEntry(extensionHFT, Function2SEL)))
```

This macro defines the symbols ExtensionHFTFunction1 and ExtensionHFTFunction2, which are the names of HFT methods.

FSExtensionHFTMgrGetEntry (extensionHFT, Function1SEL) is a method pointer to obtain the Index of Function1SEL in extensionHFT through the FSExtensionHFTMgrGetEntry method, and through Function1SELPROTO casts the pointer to the correct type. The end result is that this method can:

- ExtensionHFTFunction1();
- ExtensionHFTFunction2(3);

HFT method names and the implementation method names must be different to avoid conflict between the #define statement and the corresponding method name.

Creating a new extension HFT

The Plug-in must create its own extension HFT in order to centrally manage the method set of the Plug-in.

The HFT can be created through the FSExtensionHFTMgrNewHFT method, which requires a FS_INT32 parameter, which represents the capacity of the new HFT.

Example: Create a new extension HFT

```
FS_HFT extensionHFT = FSExtensionHFTMgrNewHFT(ExtensionMethodsNum);
```

Tips: ExtensionMethodsNum is the number of method interfaces defined by the macro in methodsCalls.h, that is, the size of the capacity that HFT needs to set.

Adding the HFT to the host environment

After creating your own extension HFT, you need to add it to the host environment before it can be called by other Plug-ins.

You can add HFT through the **FSExtensionHFTMgrAddHFT** method, which has the following parameters：

- A string pointer represents the Plug-in HFT name
- A FS_INT32 value, which represents the Plug-in HFT version
- The HFT to be added

Example: Add the HFT to the host environment

```
FSExtensionHFTMgrAddHFT(EXTENSION_HFT_NAME, EXTENSION_HFT_VERSION, extensionHFT);
```

Tips:

- In step "Create a new extension HFT", we create the HFT which is named extensionHFT.
- EXTENSION_HFT_NAME is the name of HFT defined by the macro in methodsCalls.h.
- EXTENSION_HFT_VERSION is the version of HFT defined by the macro in methodsCalls.h.

Adding the address of the methods to the extension HFT

The methods address provided by the Plug-in needs to be added to the newly created HFT so that the method provided by the Plug-in can be called by other Plug-ins.

The methods address can be added to and from the HFT through the **FSExtensionHFTMgrReplaceEntry** method. This method has the following parameters.

- An HFT object to add the method address
- A FS_INT32 value, which represents the index of the adding method in HFT
- The address of the method to be added.

Example: Add the address of the methods to the extension HFT

```
FSExtensionHFTMgrReplaceEntry(extensionHFT, Function1SEL, &Function1);
FSExtensionHFTMgrReplaceEntry(extensionHFT, Function2SEL, &Function2);
```

Tips:

- In step "Creating HFT methods", we create HFT methods named Function1 and Function2.
- In step "Create a new extension HFT", we create the HFT which is named extensionHFT.

- Function1SEL and Function2SEL are the index of HFT methods defined by the macro in methodsCalls.h.

Importing an existing HFT

You must import an existing HFT to invoke methods that are exposed through the HFT. To import an existing HFT, you must invoke the FSExtensionHFTMgrGetHFT method within the **PIImportReplaceAndRegister** handshaking method.

The FSExtensionHFTMgrGetHFT method requires the following arguments:

- A string pointer represents the Plug-in HFT name
- A FS_INT32 value, which represents the Plug-in HFT version

Example: Importing an existing HFT

```
#include "methodsCalls.h"

FS_HFT extensionHFT = NULL;

FS_BOOL PIImportReplaceAndRegister(void)
{
	extensionHFT = FSExtensionHFTMgrGetHFT(EXTENSION_HFT_NAME, EXTENSION_HFT_VERSION);
	return TRUE;
}
```

Tips: In step "Adding the HFT to the host environment", we added the HFT with the name EXTENSION_HFT_NAME and the version number EXTENSION_HFT_VERSION.

Invoking HFT methods

After you import an HFT, you can invoke a method that it has made available. For example, after you import the MyHFT HFT, you can invoke the following methods:

- ExtensionHFTFunction1
- ExtensionHFTFunction2

However, you must include the header file that defines the HFT method name in the source file in which an HFT method is invoked. Because the above methods are declared in a header file named methodsCalls.h, you must specify the following statement to successfully invoke these methods:

```
#include "methodsCalls.h"
```

If you do not include the appropriate header file, you will receive a compile error.

Example: Invoking HFT methods

```
#include "methodsCalls.h"

ExtensionHFTFunction1();
ExtensionHFTFunction2(3);
```

Tips:

- In step "Importing an existing HFT", we get the HFT which is named extensionHFT.

Global plug-in

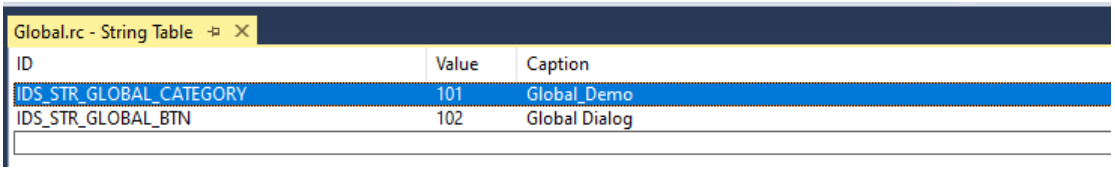
This chapter will describe how to globalize dialog, ribbon button in plugin. As we know, in application scenario, our plug-in functions need to support multiple languages and be used by customers in various countries.

By following our steps, you will be able to globalize your plugin.

Globalize category and ribbonbutton on Windows

Define resource file

On windows platform, we can use resource file to define some string within EN values. Like:



Load string by specified ID

Then use FRLanguageLoadString to load string by specified ID, it will get text by current language ID.

Example: Load string by specified ID

```
std::wstring CStarterApp::LoadStringFromID(int nID)
{
   wchar_t *pStringBuf = NULL;
   FS_INT32 nLen = 0;
   // get text by specified ID
   nLen = FRLanguageLoadString(g_pLanguage, nID, pStringBuf, nLen);
   pStringBuf = new wchar_t[nLen + 1];
   memset(pStringBuf, 0, sizeof(wchar_t) * (nLen + 1));
   nLen = FRLanguageLoadString(g_pLanguage, nID, pStringBuf, nLen + 1);
   std::wstring wsStr = pStringBuf;
   delete[]pStringBuf;
   return wsStr;
}
```

Set text by LoadStringFromID

Next, Set category or button text by LoadStringFromID.

Example: Set text by LoadStringFromID

```
FR_RibbonBar fr_Bar = FRAppGetRibbonBar(NULL);

wstring wsCategoryName = theApp.LoadStringFromID(IDS_STR_GLOBAL_CATEGORY);
FR_RibbonCategory fr_Category = FRRibbonBarAddCategory(fr_Bar, "Global_Demo", wsCategoryName.c_str());
FS_COLORREF m_clrBaseBorder = RGB(222, 222, 222);
FR_RibbonPanel fr_Panel = FRRibbonCategoryAddPanel(fr_Category, "RibbonPanel_1",
(FS_LPCWSTR)L"RibbonPanel_1", NULL);

//Create a Ribbon button
FS_INT32 nElementCount = FRRibbonPanelGetElementCount(fr_Panel);

wstring wsBtnName = theApp.LoadStringFromID(IDS_STR_GLOBAL_BTN);
//set text by
FR_RibbonButton fr_Button = (FR_RibbonButton)FRRibbonPanelAddElement(fr_Panel, FR_RIBBON_BUTTON,
      "GlobalDialog", (FS_LPCWSTR)wsBtnName.c_str(), nElementCount);

FR_RibbonElement fr_ElementButton = FRRibbonPanelGetElementByName(fr_Panel, "GlobalDialog");
FRRibbonElementSetExecuteProc(fr_ElementButton, GlobalDlgExecuteProc);
```

Prepare xml for language text

Prepare a language xml file, it named with pluginname+"_"+language.xml.
If my plugin name is Global, and it need to globalize to zh-CN and zh-TW, then we will prepare Globallang_zh-CN.xml and Globallang_zh-TW.xml. And put them in such path:

```
├──lang
|   ├──zh-CN
|   |     Globallang_zh-CN.xml
|   |
|   └──zh-TW
|         Globallang_zh-TW.xml
|
└──plugins
```

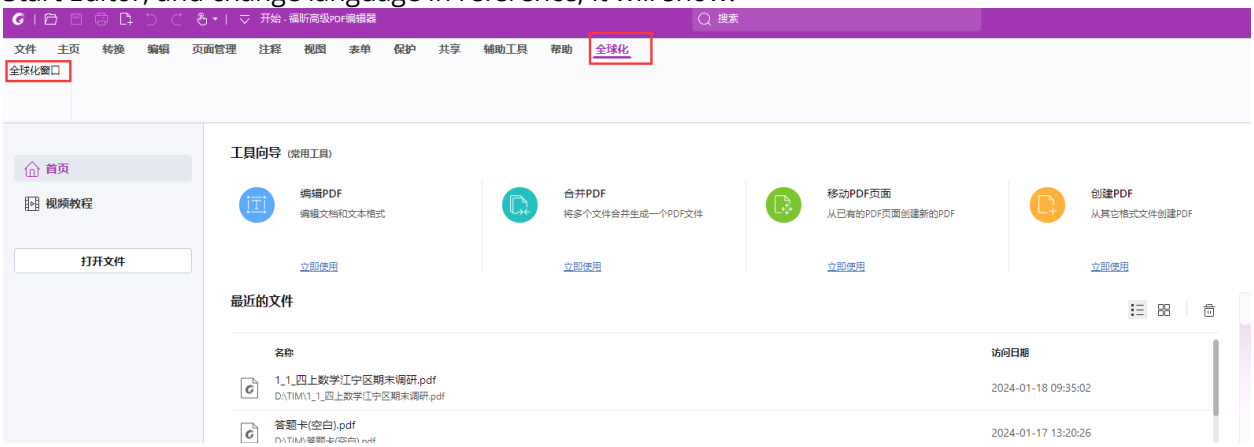
Global.fpi

InstallGlobal.xml

And the content of them is like this:

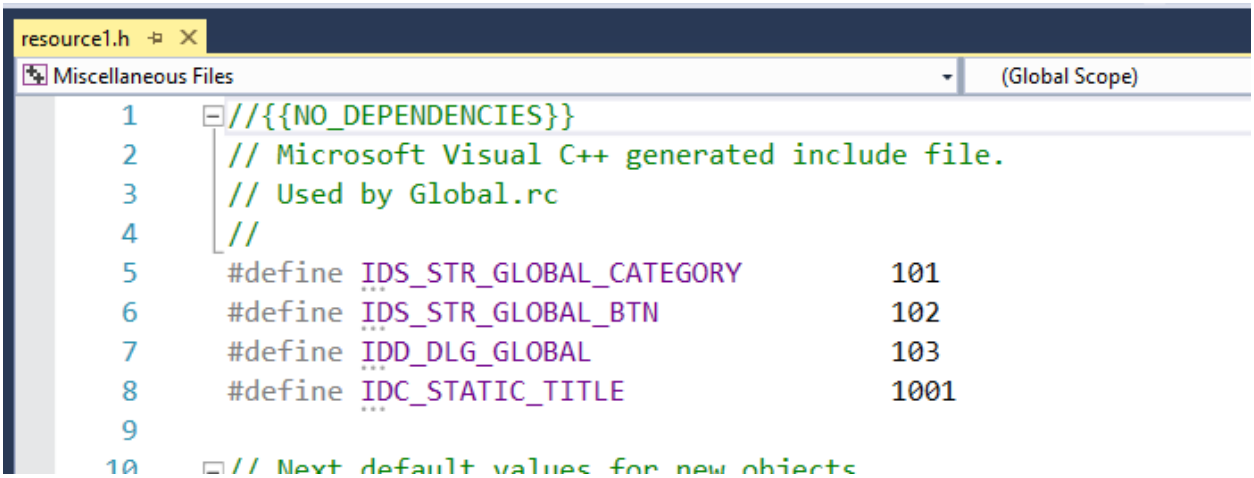
```
<?xml version="1.0" encoding="UTF-8"?>
<Reader version="12.1.0.0322" lastdate="2023//06//17" language="zh-CN" publisher="Foxit">
<Font>
<PreFont FaceName="Segoe UI" Charset="0"/>
</Font>
<Strings>
<string id="101" text="全球化"/>
<string id="102" text="全球化窗口"/>
</Strings>
</Reader>
```

Start Editor, and change language in reference, it will show:



Globalize dialog on Windows

As we know, when we add a dialog, it will list the ID of the dialog and the controls in the dialog in resource.h.



Prepare a language xml file, it named with pluginname+"_"+language.xml.

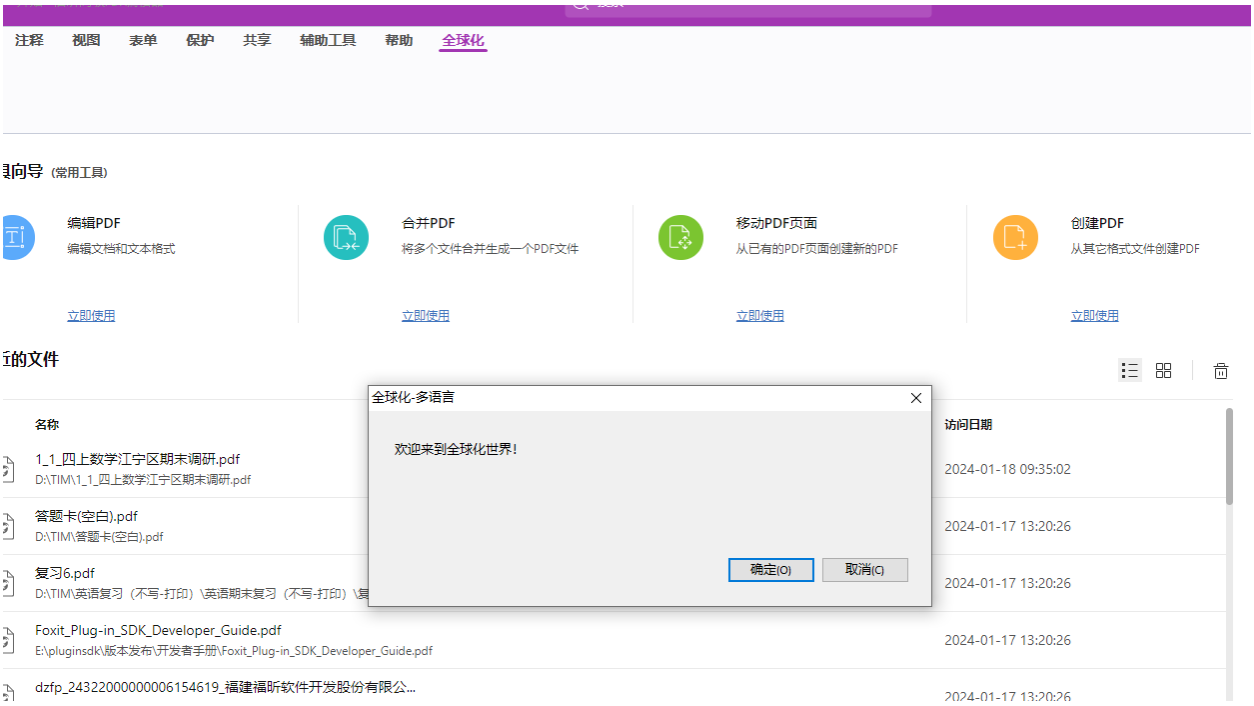
If my plugin name is Global, and it need to globalize to zh-CN and zh-TW, then we will prepare Globallang_zh-CN.xml and Globallang_zh-TW.xml. And put them in such path:

```
├──lang
│   ├──zh-CN
│   │      Globallang_zh-CN.xml
│   │
│   └──zh-TW
│          Globallang_zh-TW.xml
│
└──plugins
     Global.fpi
     InstallGlobal.xml
```

And the content of Globallang_zh-CN.xml is like this:

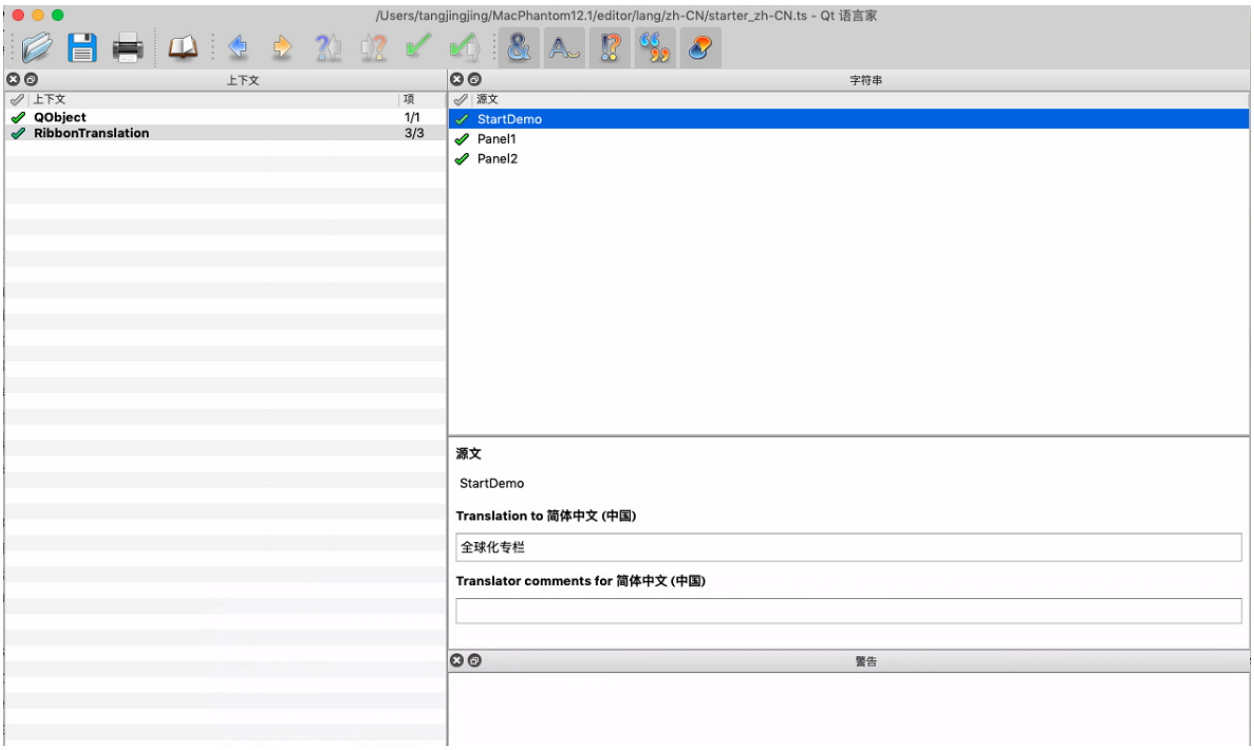
```
<?xml version="1.0" encoding="UTF-8"?>
<Reader version="12.1.0.0322" lastdate="2023//06//17" language="zh-CN" publisher="Foxit">
<Font>
<PreFont FaceName="Segoe UI" Charset="0"/>
</Font>
<Dialogs>
<Dialog id="103" text="全球化-多语言" left="0" top="0" width="569" height="224">
<dlgitem id="1001" text="欢迎来到全球化世界！" left="26" top="32" width="501" height="109"/>
<dlgitem id="1" text="确定(&amp;O)" left="359" top="146" width="88" height="26"/>
<dlgitem id="2" text="取消(&amp;C)" left="453" top="146" width="88" height="26"/>
</Dialog>
</Dialogs>
<Version>
<VerDes name="FileDescription" value="该插件用于管理所有授权。" />
<VerLeg name="LegalCopyright" value="2016-2023 福昕软件 版权所有。" />
</Version>
</Reader>
```

Start Editor, and change language in reference, it will show:



Globalize dialog on Mac

On mac platform, we just use common globalization solutions. We can use Linguist to translate text, use lupdate to generate .ts file, and then use lrelease to generate .qm file.



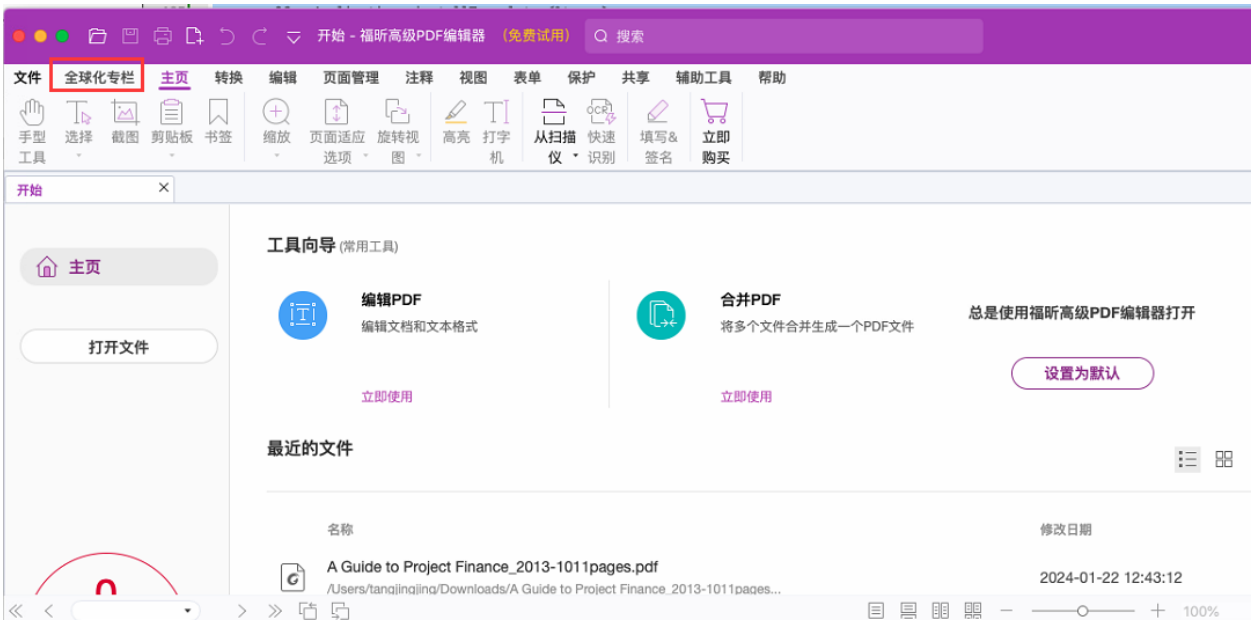
And we load the specified .qm file by current language, then install the translator.

Example: Get global text by specified .qm file

```
FS_WideString langName = FSWideStringNew();
FRLanguageGetLocalLangName(&langName);
FS_LPCWSTR name = FSWideStringCastToLPCWSTR(langName);
QTranslator trans;
if(wcscmp(name, L"zh-CN") == 0)
{
  trans.load("/tmp/MacPhantom12.1/editor/lang/zh-CN/starter_zh-CN.qm");
  QCoreApplication::installTranslator(&trans);
}

FR_RibbonCategory hRibbonCategory =
FRRibbonBarGetCategoryByName(hRibbonBar,  "Ribbon_Category_Demo");
if(hRibbonCategory == NULL)
{
  QString categoryName = QCoreApplication::translate("RibbonTranslation", "StartDemo");
  std::wstring ws = categoryName.toStdWString();
  FS_LPCWSTR categoryIps = ws.c_str();
  hRibbonCategory = FRRibbonBarAddCategory(hRibbonBar,  categoryName.toUtf8(), categoryIps);
}
```

And the result is like this:


开始 - 福昕高级PDF编辑器 （免费试用）
搜索
文件
全球化专栏
主页
转换
编辑
页面管理
注释
视图
表单
保护
共享
辅助工具
帮助
手型工具
选择
截图
剪贴板
书签
缩放
页面适应选项
旋转视图
高亮
打字机
从扫描仪
快速识别
填写&签名
立即购买
开始
主页
打开文件
工具向导 (常用工具)
编辑PDF
编辑文档和文本格式
立即使用
合并PDF
将多个文件合并生成一个PDF文件
立即使用
总是使用福昕高级PDF编辑器打开
设置为默认
最近的文件
名称
修改日期
A Guide to Project Finance_2013-1011pages.pdf
/Users/tangjingjing/Downloads/A Guide to Project Finance_2013-1011pages...
2024-01-22 12:43:12
100%

Getting started with the samples

Foxit Plug-in SDK provides some samples for Windows and Mac OS platforms in the "samples" folder.

Samples Introduction

Starter

Starter Sample contains the minimum functionality each plugin must implement in order to communicate with Foxit PDF Reader/Editor. The main files for this sample are "Starter.cpp" and "PIMain.cpp". (see: Creating Plugin).

The relevant section of code starts with the compiler directive extern “C” {...}. The relevant section of code starts with the compiler directive extern “C” {...}. Note that this directive encapsulates several functions:

- PlugInMain
- PISetupSDK
- PIHandshake
- PIExportHFTs
- PIImportReplaceAndRegister
- PILoadMenuBarUI
- PIReleaseMenuBarUI
- PILoadToolBarUI
- PIReleaseToolBarUI
- PILoadRibbonUI
- PILoadStatusBarUI
- PIInitData
- PIUnload

These functions make up a “C style” interface (hence the compiler directive) that Foxit PDF Reader/Editor uses during plugin initialization. Every plugin must maintain this interface. The initialization procedure, or “handshaking”, is handled by the PlugInMain function, which is the main entry point of plugin. The following functions in the “Starter” \ have default implementations that provide Foxit PDF Reader/Editor with pointers to user defined plugin specific functions:

- PlugInMain
- PISetupSDK
- PIHandshake

It is the developer's responsibility to add custom application logic to these plugin specific functions. Since the "Starter" plugin serves as a skeleton plugin application, these functions are left blank. For example, if the "Starter" demo allocated any memory, it would need to release that memory when the user closes Foxit PDF Reader/Editor. When Foxit PDF Reader/Editor is closed, the PIUnload() function is invoked. The developer would add code in PIUnload() to de-allocate the memory. Plug-ins must define a pointer reference to Core HFT Manager and a pointer to receive the Plug-in unique ID.

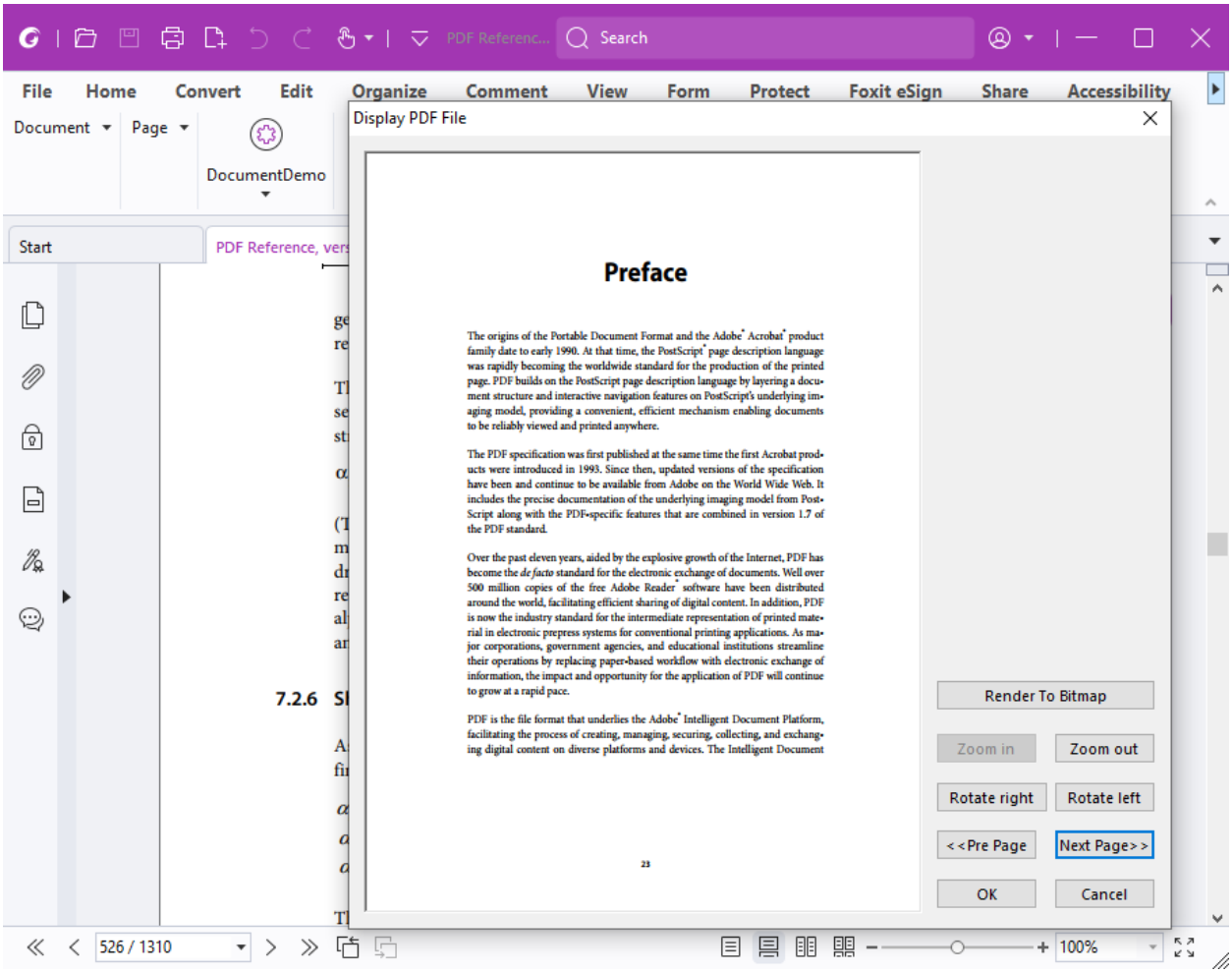
Document

Document sample demonstrates how to operate PDF Document by Foxit Plugin SDK. (See: Working the Documents)

It contains the following functions:

- Open, save, print and close PDF Documents.
- Get/Set the Permission of PDF Documents.
- Insert, replace, extract the PDF Pages from PDF Documents.
- Convert PDF Documents to the other format.
- Get the number of document pages, reload the page and refresh the page view.
- Set the PDF object as the current object.
- Add the PDF Form objects.

Screenshot:



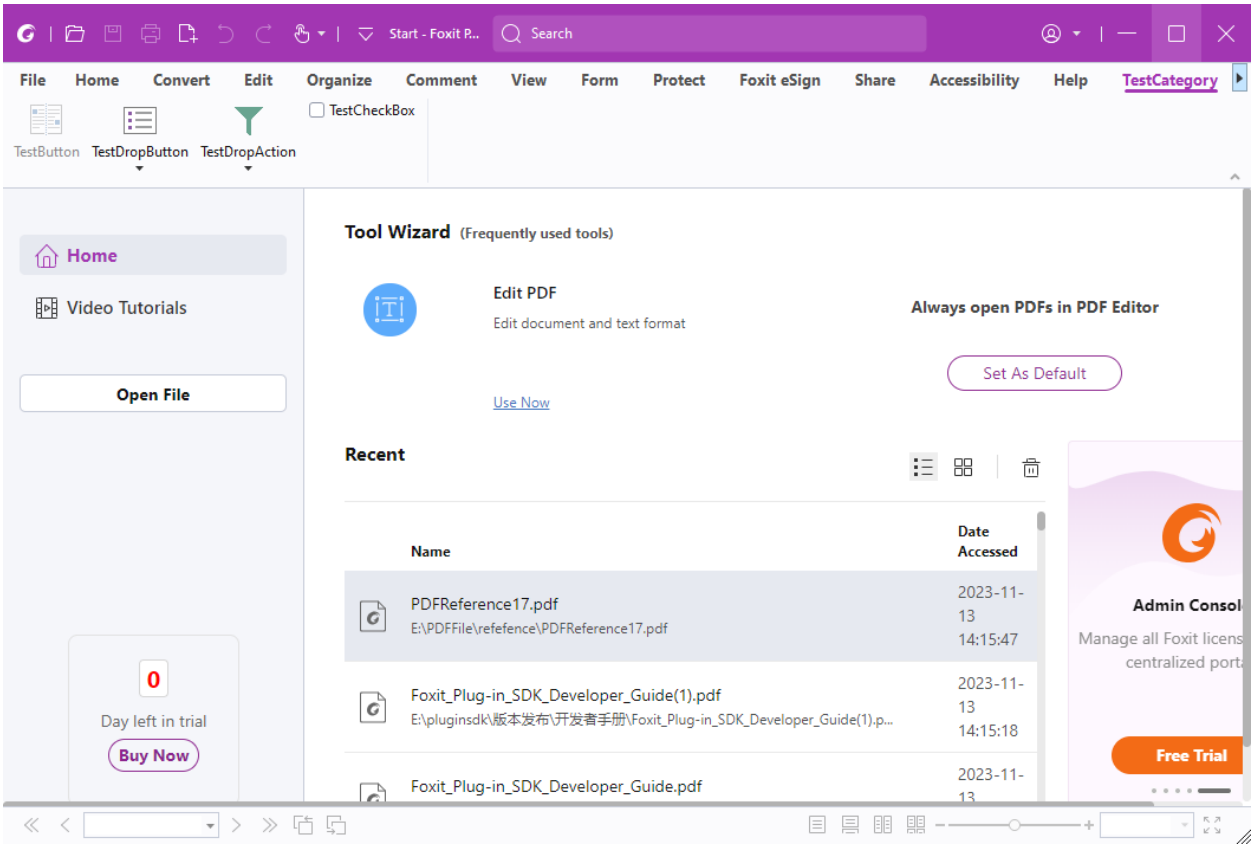
Ribbon bar

Ribbon bar sample demonstrates how to interact with Ribbon bar in Foxit PDF Reader/Editor. (See: Ribbon bar and buttons)

It contains the following functions:

- Create new Ribbon Category.
- Create new Ribbon Panel.
- Create new Ribbon Elements.
- Retrieve Ribbon Categories and Ribbon Panels.
- Create button callback functions.

Screenshot:

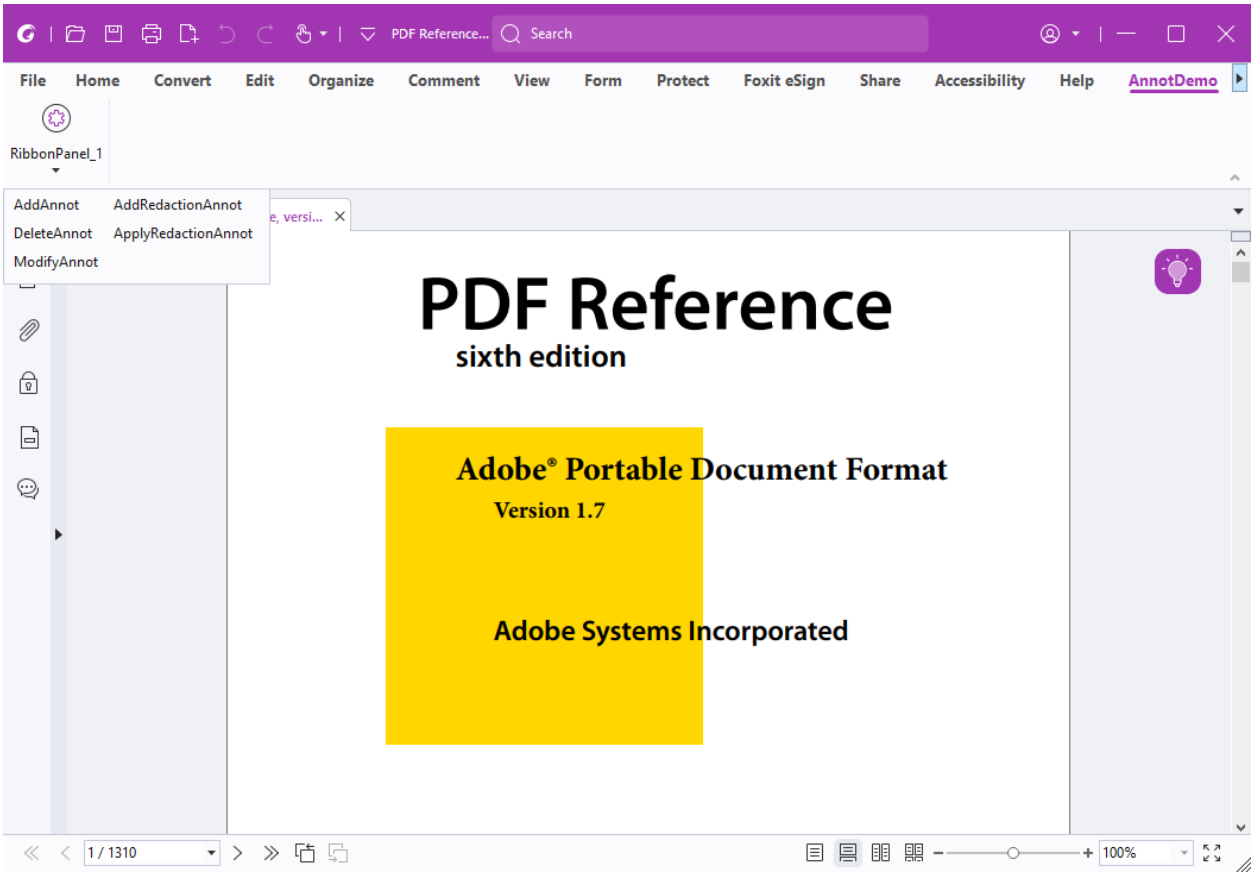


Annotations

Annotations sample demonstrates how to interact with PDF annotations. (See: working with Annotations)

- Create markup annotations.
- Retrieve the exist annotations.
- Delete annotations.
- Work with markup panel.
- Work with redaction annotations.

Screenshot:

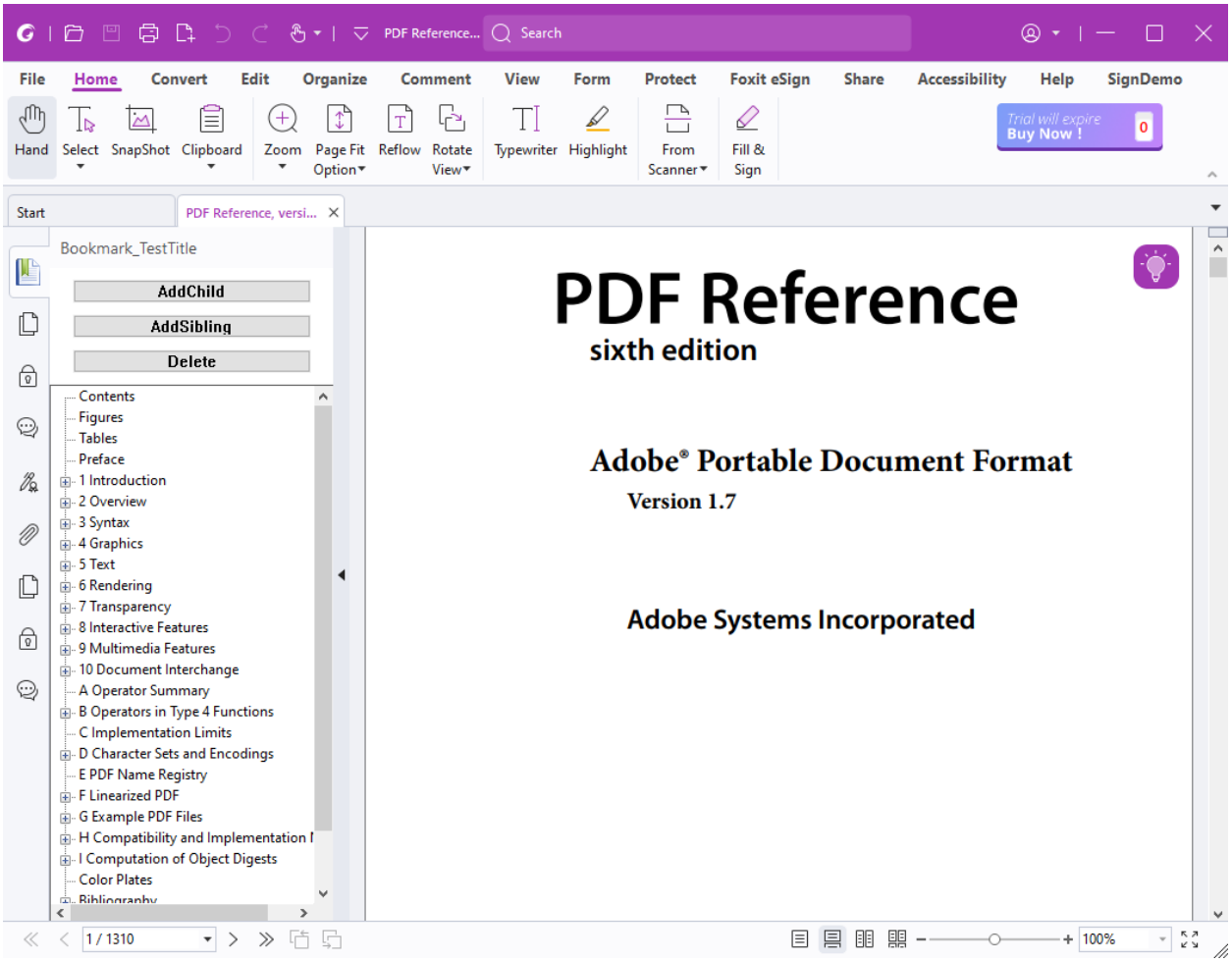


Bookmark

Bookmark sample demonstrates how to interact with PDF bookmark. (See: working with bookmark)

- Create new bookmarks.
- Retrieve bookmarks.
- Delete bookmarks.
- Activate bookmarks.

Screenshot:

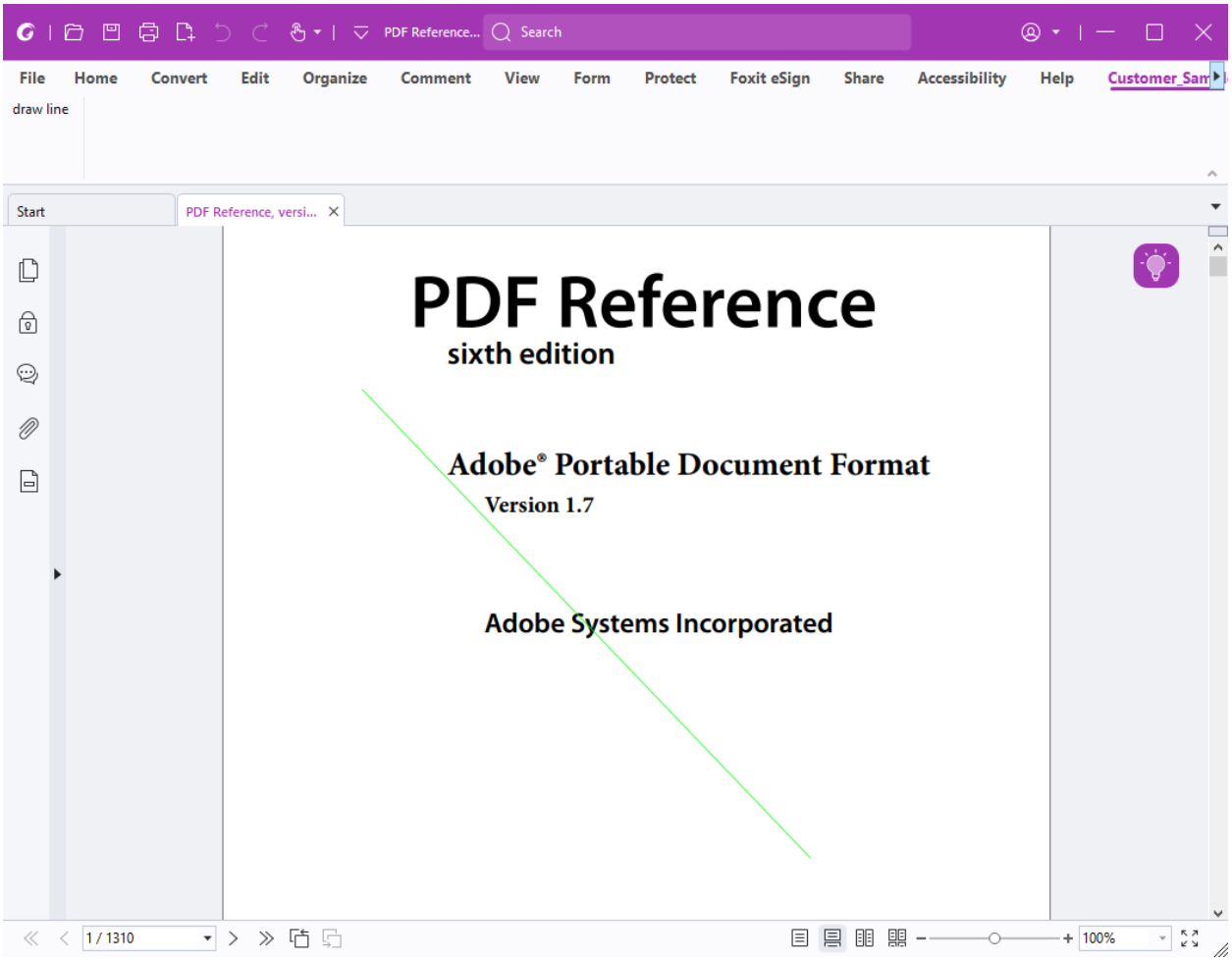


Custom Tool

Custom tool sample demonstrates how to create a custom tool in Foxit PDF Reader/Editor.

Tool is an object that can handle key presses and mouse events in the region of document view. Tools do not handle mouse events in other parts of Foxit window, such as navigation pane. At any time, there is only one active tool. Foxit PDF Reader/Editor has some built-in tools, such as Hand tool, Zoom tool, Link tool. You can get other built-in tools by Foxit Plugin SDK, and you can add a new customized tool to Foxit PDF Reader/Editor.

Screenshot:

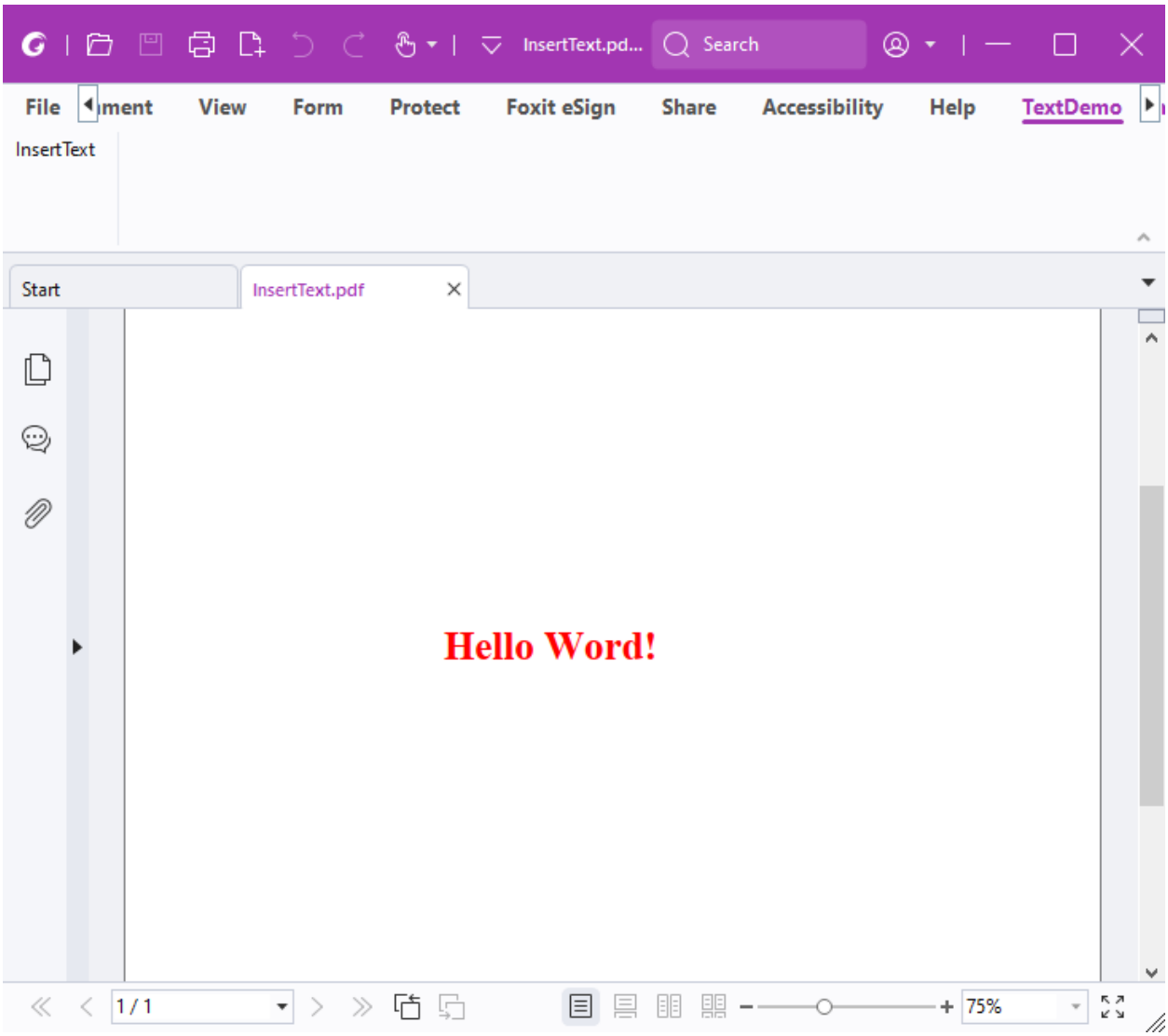


Insert Text

Insert Text sample demonstrates how to add new Text Object into PDF document. (See: Inserting Text into PDF Documents)

- Create a new blank PDF document.
- Create a new blank PDF Page.
- Create the dictionary of PDF text object.
- Set the properties of text object.
- Insert text object into PDF page.
- Refresh the content stream of the text object.
- Save the PDF document.

Screenshot



Extension HFT

A Host Function Table (HFT) is a mechanism for managing the Foxit Plug-in SDK methods. It is implemented as a pointer array that stores the addresses of Plug-in SDK methods. The methods are grouped together based on the types of objects they are associated with. Each group of methods has a specific HFT for performing actions on a specific object type. All of these HFTs are managed by the Core HFT manager. The manager indexes the HFTs by category.

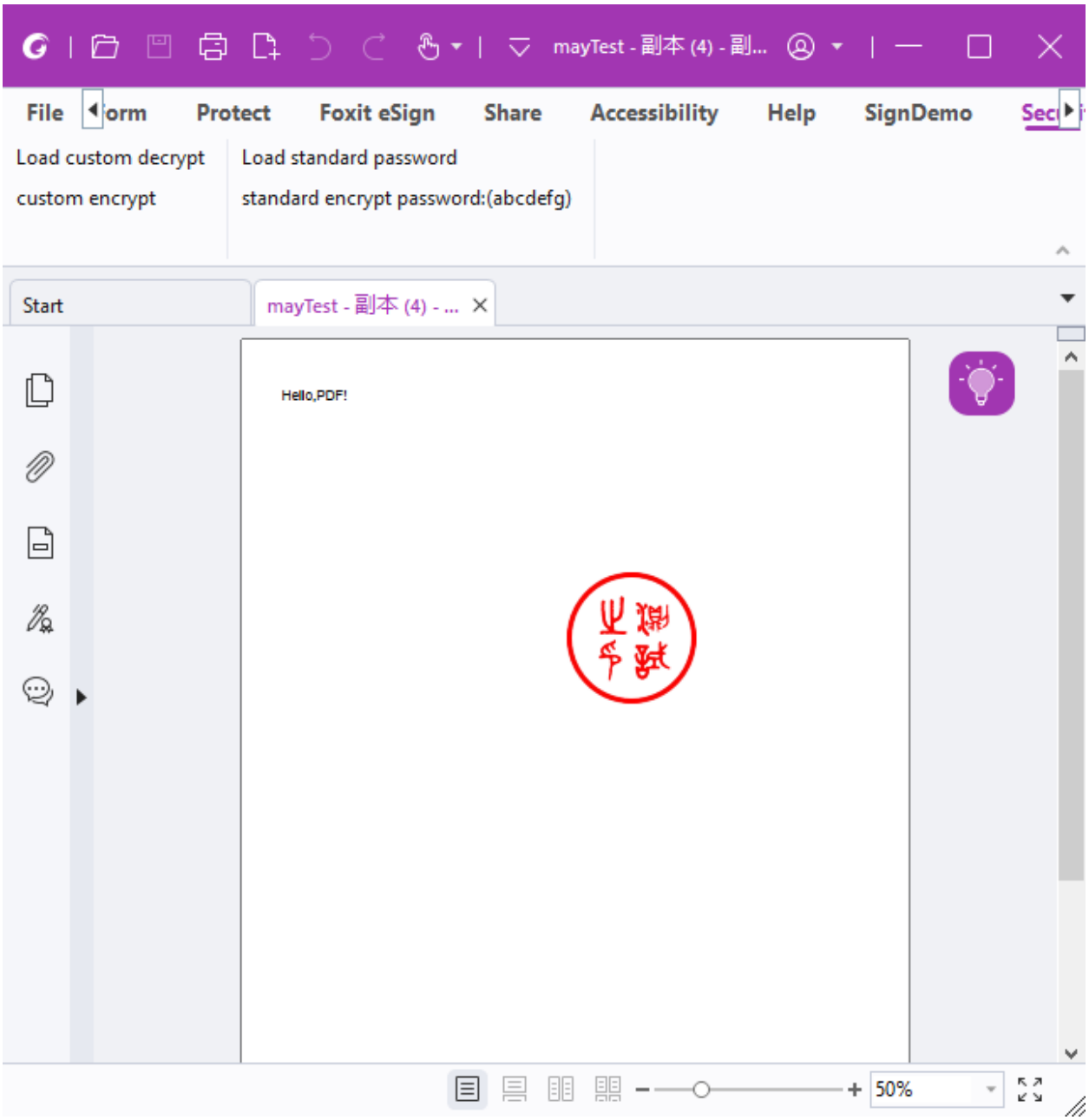
Extension HFT sample demonstrates how to work with the Host Function Table in your plugins. (See: Working with Host Function Tables):

- Export host function tables
- Import an existing HFT
- Invoke HFT methods

Security

Security sample demonstrates how to secure PDF documents. Foxit Plug-in SDK API provides a range of encryption and decryption functions to meet different level of document security protection. Users can use regular password encryption and certificate-driven encryption, or using their own algorithm for custom security implementation.

Screenshot:


mayTest - 副本 (4) - 副...
File
orm
Protect
Foxit eSign
Share
Accessibility
Help
SignDemo
Load custom decrypt
custom encrypt
Load standard password
standard encrypt password:(abcdefg)
Start
mayTest - 副本 (4) - ...
Hello,PDF!
50%

Running the samples using Visual Studio

Before running the samples, please make sure that you have read the chapter "Creating Plugin".

1. Copy your own cert.txt to res directory of the samples respectively to replace the default one.
2. Copy your own frdpisdkey.txt to /Plugin-sdk/tools directory.
3. Open the project with Visual Studio 2022.
4. Compile and generate fpi file to the lib/plugins directory which is peer dictionary of Plug-in SDK directory.
5. Copy XML file to FPI file directory.
6. Launch Foxit PDF Reader/Editor and click Help → Foxit Plugins button.
7. Click "Install plugin" and choose XML file from step 5.
8. Plugin installed successfully.

Running the samples using Qt

1. Copy your own cert.txt to res directory of the samples respectively to replace the default one.
2. Copy your own frdpisdkey.txt to /Plugin-sdk directory.
3. Add cert.txt to Qt project resource and make sure that callback functions and PIAuthorize can read this file.
4. Compile and generate fpi file to the lib/fxplugins directory which is peer directory of Plug-in SDK directory.
5. Run the command line for signing the plugin in Plugin-sdk/tools directory. Example:

```
./PiSignatureGen -c ../samples/starter/cert.txt -k ../samples/starter/frdpisdkey.txt -
p ../../../lib/fxplugins/libstarter.dylib
```

6. Launch Foxit PDF Reader/Editor and click Help → Foxit Plugins button.
7. Click "Install plugin" and choose XML file from step 5.
8. Plug in installed successfully.