## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Terminology</td>
<td>3</td>
</tr>
<tr>
<td>Linguistic Library Design at a Glance</td>
<td>4</td>
</tr>
<tr>
<td>Installing Linguistic Library and Plug-ins</td>
<td>5</td>
</tr>
<tr>
<td>Exercising Adobe Linguistic Library</td>
<td>6</td>
</tr>
<tr>
<td>Architecture</td>
<td>7</td>
</tr>
<tr>
<td>Plug-in Interface Specification.</td>
<td>8</td>
</tr>
<tr>
<td>Plug-in APIs</td>
<td>10</td>
</tr>
<tr>
<td>Run-time Architecture</td>
<td>12</td>
</tr>
<tr>
<td>Working with Linguistic Library SDK Samples</td>
<td>16</td>
</tr>
<tr>
<td>Content of SDK Samples</td>
<td>16</td>
</tr>
<tr>
<td>Plug-in Interface Implementations</td>
<td>18</td>
</tr>
<tr>
<td>Compile and Test SampleProvider</td>
<td>19</td>
</tr>
<tr>
<td>Build your First Linguistic Library Plug-in Using SampleProvider as a Template</td>
<td>20</td>
</tr>
<tr>
<td>Linguistic Library Plug-in API Reference</td>
<td>22</td>
</tr>
<tr>
<td>LinguisticPlugin.h</td>
<td>22</td>
</tr>
<tr>
<td>ILiloProviderPlugIn Interface</td>
<td>23</td>
</tr>
<tr>
<td>ILiloSpellChecker interface</td>
<td>24</td>
</tr>
<tr>
<td>ILiloHyphenation Interface</td>
<td>25</td>
</tr>
<tr>
<td>ILiloUserDictionary Interface</td>
<td>26</td>
</tr>
</tbody>
</table>
Introduction

This programming guide will help you develop plug-ins for Adobe® Linguistic Library. Linguistic Library makes it possible to add linguistic services like spell checkers and hyphenation and User Dictionary to Adobe products, through the Linguistic Library API.

Linguistic Library is not a shrink-wrapped product. Instead, it provides common linguistic services for many Adobe products; thus, it is an important component of these products.

Linguistic Library has an extensible architecture. You can extend the functionality of Linguistic Library to provide customized linguistic services by implementing plug-ins for it. The advantage of implementing plug-ins through Linguistic Library is that these additional linguistic services are available to all Adobe products automatically if they use Linguistic Library. To provide services through individual product extensions; you would have to write programs against every product API.

The intended audience for this document is developers with a basic understanding of Microsoft Component Object Model (COM) technology and Apple’s CFPlug-in API (since the Linguistic Library plug-in architecture is based on these technologies).

Terminology

The following terms are used in this document:

- **Client application** — An Adobe application that uses linguistics services provided by Linguistic Library.

- **LILO** — Shorthand for Adobe Linguistic Library.

- **Plug-in** — Except where explicitly specified, “plug-ins” in this document refers to Linguistic Library plug-ins that extend Linguistic Library and provide additional, customized linguistic services.

- **<Plug-in folder>** — The location of the Linguistic Library plug-ins folder, On Mac OS®, it is /Library/Application Support/Adobe/Linguistics/6.0/Providers/Plugins2. On Windows®, it is %COMMONPROGRAMFILES%\Adobe\Linguistics\6.0\Providers\Plugins2\.

- **<SDK>** — The path where you installed the Linguistics Library SDK.

- **Shell window** — The command-line window. On Mac OS, use the Terminal utility (located in /Applications/Utilities). On Windows, use Command Prompt (located in the Accessories folder from the Start menu).
Linguistic Library Design at a Glance

Linguistic library is a shared library that provides a uniform API to Adobe applications that need linguistic services like spelling, hyphenation, user dictionary, and thesaurus. These client applications include Adobe Acrobat®, LiveCycle® Form Designer, Adobe InDesign® CS6, Adobe Illustrator® CS6 and Adobe Photoshop® CS6.

The latest Linguistic Library, shipped with Adobe Creative Suite® 6 applications, is version 6. The filename of the library is AdobeLinguistics.dll on Windows and AdobeLinguistics.framework on Mac OS. They are located in the same directory of the application executables and are installed by application installers.

Linguistic Library behaves as an intermediary between Adobe applications and linguistic service providers. It provides a uniform API to Adobe applications, so changes and additions to the linguistic service do not propagate to the application. On the other hand, it provides an open architecture that allows third-party linguistic service providers to supply linguistic services through the Linguistic Library, so the services can be used by all Adobe applications. The independence achieved by the Linguistic Library makes it possible for third-party providers to add new linguistic services even after the Adobe applications are installed.

Figure 1 illustrates the relationships between Adobe applications, Linguistic Library, and linguistic service providers (through Linguistic Library plug-ins).

**Figure 1: Relationships between Applications, Linguistic Library, and Plug-ins**

The interface between Adobe applications and the Linguistic Library is intended for Adobe internal use and not covered in this document.

The interface between the Linguistic Library and linguistic service providers is the plug-in architecture. COM was chosen for the plug-in specification, and we recommend the CFPlug-in for writing plug-ins on Mac OS. For details, see the “Adobe Linguistic Library Plug-in Architecture” chapter.

Every Linguistic Library plug-in should have its own dictionary data. It can specify the languages and services it supports. For example, if a custom Linguistic Library plug-in
supports US English and provides a spell-checking service, the end user of all Adobe applications (like InDesign) can choose the custom spell-checking service in US English.

**NOTE:** Although the Linguistic Library is designed to support all linguistic services, in version 6.0, only interfaces for spelling, hyphenation and User Dictionary services are provided in the SDK.

Another benefit of the Linguistic Library is that it supports plug-in versioning. Plug-ins are installed under per-version folders and can co-exist with each other. An individual application can select the specific Linguistic Library plug-in version numbers it wants to use. For example, Application A might want to use MyLiloPlug-in Version 1.0, while Application B might want to use version 1.2. See Figure 2.

**FIGURE 2 Linguistic Library Plugin Versioning**

---

**Installing Linguistic Library and Plug-ins**

Linguistic Library (AdobeLinguistics.dll on Windows, AdobeLinguistics.framework on Mac OS) is a shared library and is installed in the same location as the application executables (Windows) or linked into the application (Mac OS) during Adobe application installation. The Linguistic Library’s own data file and any Linguistic Library plug-ins that ship with Adobe applications are also installed during the application installation process, but to the common Linguistic Library data and plug-in folder. See Table 1 for recommended and default folders where these Linguistic Library components should be installed for CS6 applications.

**Note:** Adobe Linguistic Library 6.0 does not support the Version 1.0 Plug-ins.
### TABLE 1 Linguistic Library Installation Locations

<table>
<thead>
<tr>
<th>Component</th>
<th>Windows</th>
<th>Mac OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linguistic</td>
<td>AdobeLinguistics.dll at the same location as the application. For example C:\Program Files\Adobe\Adobe InDesign CS6\ AdobeLinguistic.dll.</td>
<td>Under the application’s package related to executable_path/../Frameworks. For example, /Applications/Adobe InDesign CS6/Adobe InDesign CS6.app/Contents/Frameworks/AdobeLinguistic.framework /Versions/3/AdobeLinguistic</td>
</tr>
<tr>
<td>library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data file</td>
<td>Common Program Files. For example %COMMONPROGRAMFILES%\Adobe\Linguistics\6.0\LanguageNames2\DisplayLanguageNames.en_US.txt</td>
<td>Common application-support Folder. For example, /Library/Application Support/Adobe Linguistics/6.0 /LanguageNames2\DisplayLanguageNames.en_US.txt</td>
</tr>
<tr>
<td>Plugins</td>
<td>Common Linguistic Library plug-ins folder. For example, %COMMONPROGRAMFILES%\Adobe\Linguistics\6.0\Providers\Plugins2\WRLiloPlugin1.3</td>
<td>Common Linguistic Library plug-ins folder. For example /Library/Application Support/Adobe/Linguistics/6.0/Providers/Plugins2/WRLiloPlugin1.3.bundle</td>
</tr>
</tbody>
</table>

**NOTE:** (Windows only) To install custom Linguistic Library plug-ins after an application is installed, you need to register with the systems in addition to copying plugins to the specified folder. In the command shell, execute regsvr32 <plug-in DLL name> (For regsvr32 to work correctly, you must be logged in as the administrator).

**NOTE:** Acrobat and Acrobat Reader install Linguistic library to different locations.

### Exercising Adobe Linguistic Library

The Linguistic Library itself does not have a user interface. You must execute an Adobe application to access the functionality and services provided by the Linguistic Library and its plug-ins. For example, to check out the spelling service provided by the Linguistic Library, follow these steps:

1. Install InDesign CS6 or another Adobe application that uses the Linguistic Library version 6. (You may install one of the Adobe CS6 suite products like InDesign that uses Linguistic library LILO.)


3. Launch InDesign.

4. (Optional) If you want the preference set in the next step to apply to only one document, open an existing document or create a new one.
5. Bring up the dictionary preference dialog, by choosing Edit > Preference > Dictionary... (Windows) or InDesign > Preference > Dictionary... (Mac OS).

6. Select a language. The Linguistic Library 6.0 (CS6) supports more than 40 languages. If your custom Linguistic Library plug-ins supports another language, the language should appear in the drop-down list. Multiple providers can support the same language; for example, your custom Linguistic Library plug-ins may also support US English.

7. Choose the spelling, hyphenation or user dictionary service provider in the selected language. If your custom Linguistic Library plug-in provides a service for the selected language, the service appears in the drop-down list for spelling, hyphenation or user dictionary. If you choose Adobe-supported languages and services, you are exercising Adobe-provided Linguistic Library functions; if you choose a service provided by your own, custom Linguistic Library plug-in, you are exercising your custom plug-in.

8. If you have not already done so, create a new document. Bring the Character panel to the front, and make sure your chosen language is shown at the bottom of the panel.

9. Create a text frame, and type a few words.

10. To test spelling, bring up the spelling dialog by choosing Edit > Spelling > Check Spelling... The dialog steps through the text with the spelling service you chose previously. Make sure it flags words that are not in the dictionary.

11. To test hyphenation, resize the text frame to see whether hyphenation works as expected.

12. To create, edit or delete dictionaries bring up the dictionary preference dialog, by choosing Edit > Preference > Dictionary... (Windows) or InDesign > Preference > Dictionary... (Mac OS). Select a provider and its language and then add existing dictionary, delete user dictionary or Create user dictionary.

Architecture

The Linguistic Library plug-in architecture is designed for third-party vendors to provide their spelling and other linguistic services to Adobe applications through the Linguistic Library’s API.

The plug-in architecture comprises the following:
- Plug-in interface specification.
- Plug-in APIs.
- Run-time architecture.
These components are discussed in the following sections.

**Plug-in Interface Specification**

The plug-in interface specification is as compatible as possible at the source level on both Windows and Mac OS.

On Windows, the standard software component architecture is COM, which defines the binary interface of a component for interoperability. COM also has guidelines for writing component software, to facilitate interface queries and dynamic linking at run time. COM has existed on Windows since 1995. It is well documented and supported fully by the Windows run-time environment.

COM offers many features, but in the Linguistic Library's architecture, we use only rudimentary features like IUnknown interface, its usage protocol, and the component registry functions to find and load Linguistic Library plug-ins.

On Mac OS, the Core Foundation library provides a plug-in architecture known as CFPlug-in that is derived from COM's IUnknown interface; therefore, it was natural to adopt COM as the plug-in binary interface on both platforms. This standardizes the calling convention at the source-code level and eliminates the need for another library to provide the plug-in architecture on Mac OS. The CFPlug-in is in the native Mach-o format, and Mac OS provides APIs for loading and basic run-time support.

Component registration on Windows differs from Mac OS, but Linguistic Library takes care of component management for applications and plug-ins. For details on the registration behavior, see “Plug-in Registration” on page 14.

**GUID in COM**

One important aspect of COM is that it uses GUID (Globally Unique IDentifier) for its class ID and interface ID. Each COM object contains exactly one class ID, or CLSID, and one interface ID (IID) for each interface it supports. Therefore, a Linguistic Library plug-in supporting the provider and the spelling interface will have three GUIDs: one CLSID and two IIDs. Usually, the CLSID and one IID are used together to locate a COM object that supports an interface.

On Mac OS, UUID (Universally Unique IDentifier) is used, although it is identical to the GUID. Instead of a CLSID, the CFPlug-in has a Type ID, and the IID is the same as on Windows.

**IUnknown**
At its most basic level, the COM interface specification is essentially a table of function pointers that resembles the vtable of an abstract class in C++. The root class that all COM classes must inherit from is the IUnknown class, defined as follows:

interface IUnknown
{
HRESULT QueryInterface(
REFIID iid,
void ** ppvObject) = 0;
ULONG AddRef(void) = 0;
ULONG Release(void) = 0;
};

On Mac OS, the IUnknown interface is defined as follows:

interface IUnknown
{
virtual HRESULT __stdcall QueryInterface(
const IID& iid
void **ppv) = 0;
virtual ULONG __stdcall AddRef() = 0;
virtual ULONG __stdcall Release() = 0;
};

The corresponding C struct is defined as:

#define IUNKNOWN_C_GUTS
void *_reserved;
HRESULT (STDMETHODCALLTYPE *QueryInterface)
(void *thisPointer, REFIID iid, LPVOID *ppv);:
ULONG (STDMETHODCALLTYPE *AddRef)(void *thisPointer);:
ULONG (STDMETHODCALLTYPE *Release)(void *thisPointer);

The C struct could be used to define the function pointer table for IUnknown and any other interface derived from IUnknown, if the compiler does not support COM interface generation. An example is listed below:

typedef struct TestInterfaceStruct {
IUNKNOWN_C_GUTS;
void (*fooMe)( void *this, Boolean flag );
} TestInterfaceStruct;

As the C struct demonstrates, the COM interface specification basically is a simple function pointer table. As long as the plug-in follows the COM protocols for the IUnknown interface, the object behavior is standardized.

Plug-in APIs
The Linguistic Library plug-in APIs are shipped with the SDK and are located in `<SDK>\api`. The plug-in API is grouped into the following categories:

- Provider
- Spelling
- Hyphenation
- User Dictionary

You must implement the provider API, since the Linguistic Library uses it to query the plug-in on what feature it supports, such as services and languages supported. The other categories are optional. For example, if you have only spelling technology, you would implement the spelling API in your plug-in.

New categories will be created and new APIs will be published in the future. For example, thesaurus services and APIs may be added in a future release of the Linguistic Library SDK. Since the API set is based on the COM interface, it is not statically bonded and can be extended without requiring existing Linguistic Library clients to recompile. A new plug-in could support a new API yet still serve existing Linguistic Library clients, if it continues to support the existing interface.

**Memory-allocation Policy**

Whenever the function calls for a plug-in to return a word or an array of data, the plug-in needs to allocate the memory with the `SysMemAllocate()` function, which is a COM memory allocation function that ensures the safe allocation of memory. The Linguistic Library is responsible for disposing the memory allocated by the plug-in. If the Linguistic Library allocates an object and passes it to the plug-in, the Linguistic Library is still responsible for de-allocating it. The plug-in should not de-allocate these objects itself.

**COM Data Types**

The Linguistic Library plug-in API uses COM data types like BSTR, SHORT, LONG, and CHAR. They are defined in the public COM documentation, available on the Microsoft Web site.

**Error Code**

When a function succeeds, it returns S_OK. When a function fails, the plug-in is required to return a COM-style error code, such as E_OUTOFMEMORY. See the COM error code types for appropriate errors.
Table 2 summarizes supported interfaces in the Linguistic Library version 4.0. For detailed descriptions of the interfaces, including every function of each interface, see “Linguistic Library Plug-in API Reference” on page 22.

**Table 2 Interface Summary**

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IALPHyphenation</td>
<td>Provides functions for hyphenation services: gets possible hyphenation points and divides a word into two parts at a hyphenation point.</td>
</tr>
<tr>
<td>IALPProvider</td>
<td>Reads the info.plist file and tells the Linguistic Library what services it provides, what potential languages it supports for a specified service, and its name in a Unicode string. This interface allows Linguistic Library clients to implement user-interface features.</td>
</tr>
<tr>
<td>IALPSpellChecker</td>
<td>Provides functions for spelling services: checks the spelling of a word and provides a list of suggestions for a word.</td>
</tr>
<tr>
<td>IALPUUserDictionary</td>
<td>Provides functions for user dictionary services like: add words, delete words and add word pairs.</td>
</tr>
</tbody>
</table>
When the Linguistic Library starts it reads the info.plist file of the plug-ins and stores the information. Later when the Linguistic Service Like spelling, hyphenation or user dictionary is requested for the Plug-in provider it checks the common registry to see if the DLL is loaded or not. If not then it loads the Plug-in DLL.

**FIGURE 3 Linguistic Library Start-up Process**

![Linguistic Library Start-up Process Diagram](image)

The initialization is done primarily by reading information from the Information Property file (info.plist) instead of scanning and loading each plug-in, to reduce initialization time.

When a Linguistic Library client requests a linguistic service (like spelling) from a particular provider, the Linguistic Library requests a spelling interface from that provider, or the plug-in. If the plug-in supports the interface, the Linguistic Library asks it to create an instance of the interface and gives it back to the Linguistic Library client. Figure 4 illustrates the work flow.

After the Linguistic Library plug-ins are loaded, the Linguistic Library acts as an agent between applications and linguistic services. The Linguistic Library plus plug-ins is an encapsulated unit.

**FIGURE 4 Linguistic Library Request for Interface**
Figure 5 conceptual relationship between plug-in components.
Figure 5 illustrated three interfaces:

- IALPSpellChecker provides the functions the plug-in needs to support spell checking. For instance, the Linguistic Library calls IsWord() to check the spelling of a word. If the word is not spelled correctly, the Linguistic Library calls CorrectWord() to get a list of suggestions.
- IALPHyphenation specifies the functions plug-in needs to implement hyphenation.
- IALPUserDictionary provides the User Dictionary functionality. For instance, the Linguistic Library calls Addword() to add words to the dictionary.

The figure shows three plug-ins: Winsoft, Hunspell, and a third-party plug-in.

**Plug-in Registration**

After a Linguistic Library plug-in is installed, it needs to register itself so the Linguistic Library can find it at run time. The registration mechanism is same on Windows and Mac. The Linguistic Library relies on a central registry to search for a plug-in to load. The central registry is discussed in more detail in “Central Registry” on page 16. For Adobe-supplied plug-ins, the registration is done as a part of installation.

When a plug-in is installed, several pieces of information are written in the central registry to properly register the plug-in. The most important ones are as follows:

- **Plug-in location** — A plug-in usually is invoked to register itself. A CFPlug-in can choose between static or dynamic registration. The CFPlug-inDynamicRegistration determines which type of registration is required. A static registration means all the necessary information is stored in its information property file, or plist file. A dynamic registration makes the operating system load the plug-in and call its registration functions. The plist should contain the name of the function that should be called for dynamic registration. For the Linguistic Library 6.0 plug-ins, only static registrations are allowed.

- **Plug-in category** — This describes what linguistic services the plug-in supports. The Linguistic Library must quickly determine what linguistic services each plug-in supports, without having to load each plug-in at start-up time. Either the installer or the plug-in itself can write this information.

- **Optional plug-in loading hint** — The Linguistic Library's client supplies this to indicate any specific plug-in that should be loaded. This is useful to the Linguistic Library clients that have tested and certified only certain plug-ins. The plug-ins that is not on this list may be loaded by the user through a preference panel implemented by the Linguistic Library client applications. The hint consists of plug-in names that the Linguistic Library should automatically load; therefore, if a plug-in is not on the list, it is not explicitly loaded unless the Linguistic Library client asks for it to be loaded. If no hint exists, the Linguistic Library loads all the plug-ins it finds.
A plug-in carries the attributes listed in Table 3 in its information properly list, which is stored in the CFBundle.

### TABLE 3 Plug-in plist Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFPluginDynamicRegisterFunction</td>
<td>String</td>
<td>Dynamic registration not supported in Lilo6.0. The value is empty</td>
</tr>
<tr>
<td>CFPluginDynamicRegistration</td>
<td>No</td>
<td>Dynamic registration not supported in Lilo6.0.</td>
</tr>
<tr>
<td>CFPluginFactories</td>
<td>(factory UUID, string)</td>
<td>Used for static registration. String is the factory function name that creates an instance of the UUID, which represents the interface it supports.</td>
</tr>
<tr>
<td>CFPluginTypes</td>
<td>(Type UUID, array of factory UUID)</td>
<td>Used for static registration. This defines what interface the plug-in supports. Each factory UUID corresponds to an interface it supports.</td>
</tr>
<tr>
<td>CFPluginUnloadFunction</td>
<td>String</td>
<td>Function name to call to unload this plug-in.</td>
</tr>
</tbody>
</table>

These attributes defines how a CFPlug-in is registered to its host, but they do not register the category or write anything to the central registry.

### Component Category

Component category is introduced in the COM specification as a way to categorize components. As the number of COM objects increases, there is a need for a way to search for components that support a particular interface.

The Linguistic Library scans the Linguistic Library plug-in folder to find plug-ins. If a plug-in is not stored in this folder, it is not loaded. Make plug-ins are installed at a central location.

Win: %COMMONPROGRAMFILES%\Adobe\Linguistics\6.0\Providers\Plugins2 folder.
Mac: /Library/Application Support/Adobe/Linguistics/6.0/Providers/Plugins2 folder.

### Plug-in Object Creation

How does a plug-in get loaded, and how does the object get created for a requested interface? Generally, plug-in needs to support a factory function that knows how to create an instance of an interface. The Linguistic Library uses this factory function to create objects for the interfaces in which it is interested.

The Linguistic Library needs to do the work of the Component Category Manager in locating a plug-in that supports the requested interface. Once one is located, the
Linguistic Library uses CFPlug-in API to get its factory function. This function creates an instance of the interface that the Linguistic Library requests. If the interface is not supported, NULL is returned. A CFPlug-in component usually identifies its factory function in its plist file.

**Plug-in Lifecycle**

The COM interface specification requires each interface to implement a reference count. The plug-in provides a function to unload, but it is entirely up to the host to decide whether to unload the plug-in. In this case, the Linguistic Library does not unload the plug-in until the Linguistic Library client is ready to quit.

**Central Registry**

The plug-in run-time environment relies on a central registry to find plug-ins and their attributes, such as categories. The Linguistic Library looks for plug-ins in the Plug-in folder at start-up time. It looks for the component category information stored in each plug-in’s information property list and stores the category information in a memory cache. Although plug-ins with static registration are not loaded, reading the property-information list of each plug-in might take awhile.

**Working with Linguistic Library SDK Samples**

This section provides guidance on how to work with the interfaces and sample code provided in the SDK.

**Content of SDK Samples**

Enclosed with this SDK, you will find both Windows and Macintosh linguistic service providers, with projects and source code located under `<SDK>`:

- The `<SDK>/api` folder contains public-interface header/API files. This directory contains the header files that must be included in a Plugin. A plugin should extend/derive the classes defined in header files.
- The `<SDK>/api/mac` folder contains the Mac specific header files.
- The `<SDK>/SamplePlugin/build/mac` directory contains the Xcode project file to compile and build projects on MAC. Mac Binaries will also be outputted to this folder after compilation. Also, related files, including Information property file (info.plist) are in this folder.
- The `<SDK>/SamplePlugin/build/win` directory contains the VS10 project file to compile and build projects on Windows. Windows Binaries will also be outputted to this folder after compilation. Also, related files, including info.plist are in this folder.
- The `<SDK>/SamplePlugin/source` directory contains the source files. It contains two sub folder:
  a) mac : for mac specific source (.h,.cpp) files
  b) win : for windows specific source (.h,.cpp) files
Common source files are kept at the root of this directory.
- The `<SDK>/docs` folder contains the documents related to the SDK.

**Header Files**

There are six files in the “api” folder. IALPProvider.h, IALPSpellChecker.h, IALPHyphenation.h, IALPUserDictionary.h are the interfaces with which a plug-in is programmed. LinguisticConditions.h and LinguisticPlugin.h define common variables used in cross-platform implementation. Since these interfaces define the contract between the Linguistic Library and a Linguistic Library plug-in, they should not be changed. Also the Mac specific header files are also found in “api/mac” directory.

**Cross-platform Source Files**

Open either SampleProvider.sln on Windows or SampleProvider.Xcodeproj on Mac OS.

You will find the following code in both projects:

1. PluginProvider.cpp and PluginProvider.h implement the IALPProvider interface.
2. ALPluginSpellChecker.cpp and ALPluginSpellChecker.h implement the IALPSpellChecker interface.
3. IALPluginHyphenation.cpp and IALPluginHyphenation.h implement the IALPHyphenation interface.

**Windows-specific Source Files**

In addition to common files, the following files also are included in SampleProvider.sln:

1. SampleProvider.def — A module-definition file that defines the exports of the plug-in DLL.
2. SampleProvider.cpp — The main DLL source file that implements exported DLL entries like DllRegisterServer, DllUnRegisterServer and CreateInstance.

**Mac-specific Source Files**

On Mac OS, Xcode projects include files that provide the CoreFoundation equivalent of Windows COM and Automation:

1. Guiddef.h, initguid.h, objbase.h oleauto.cpp, oleauto.h, and winbase.h — These are open source files.
2. Server.cpp and Server.h — These are “glue code” that implement COM on Mac OS.

Plug-in Interface Implementations

IALPProvider

The first interface you need to implement is IALPProvider. This is the required interface that tells Linguistic Library that the plug-in provides a certain service.

PluginProvider.h and PluginProvider.cpp implement two classes:

• PluginProviderFactory — Called to create ALPluginProvider instance when the plug-in is loaded. On Windows, it is called within the DllGetClassObject function (SampleProvider.cpp). On Mac OS, the plug-in entry point is PluginFactory, which creates a PluginProviderFactory class to instantiate PluginProvider via QueryInterface.

• PluginProvider — A COM that inherits the IUnknown interface and implements its functions.

NOTE: You can have your own, customized implementation of the PluginProvider class.

IALPSpellChecker

ALPluginSpellChecker.cpp and ALPluginSpellChecker.h implement the IALPSpellChecker interface.

In addition to IUnknown methods, ALPluginSpellChecker also implements the following:

• IsWord() — Returns whether a word is included in the dictionary.

• GetSuggestionList() — Gives a list of suggestions for correcting a misspelled word.

ALPluginSpellChecker defines private-member variables that support these methods. For example, m_SupportedLanguages stores a list of languages supported, and m_Capability stores a list of spell-checker options. These language-definition and spell-checker options are defined in the LinguisticPlugin.h header file.

Only two words in the dictionary are hard-coded in this sample: “Adobe”, “Lilo” and “misunderstanding”. You might want to have a separate dictionary file to store words; thus, your declaration of m_dictionary might be different.
IALPHyphenation

ALPluginHyphenation.cpp and ALPluginHyphenation.h implement the IALPHyphenation interface.

In addition to IUnknown methods, ALPluginHyphenation also implements the following:

- GetHyphenationPoints() — Returns a list of positions where a word can or cannot be broken up.
- CutWord() — Cuts a word in two parts.

As a reference implementation, the sample has hard-coded hyphenation points for only one word, “laboratory.” You might want design your own data structures for real-world hyphenation.

IALPUserDictionary

ALPlugInUserDictionary.cpp and ALPlugInUserDictionary.h implement the IALPUserDictionary interface.

In addition to IUnknown methods, ALPlugInUserDictionary also implements the following:

- AddWord() — To add a word in the user dictionary.
- AddWords() — To Add multiple words in the user dictionary.
- EraseWord() — To erase a word from the user Dictionary.
- EraseAllWords() — To erase all the words from the user Dictionary.
- AddWordPair() — To Add a misspelled and correct word pair in the user dictionary.

Compile and Test SampleProvider

Having had an impression of the sample code, let try it out. On Windows, follow these steps:

1. Use Microsoft Visual Studio C++ 200 to open <SDK>\SamplePlugin\build\win\SampleProvider.sln.
2. Build the project. You may notice there are some warnings. Ignore these warnings.
3. The compiled plug-in named SampleProvider.dll is located in <SDK>/SamplePlugin\build\win\binaries\Win32\Debug or <SDK>/SamplePlugin\build\win\binaries\Win32\Release.

4. Follow the instructions in “Installing Linguistic Library and Plug-ins” on page 5 and “Exercising Adobe Linguistic Library” on page 6 to install a custom Linguistic Library plug-in and exercise Linguistic Library. Do not forget the step that executes the “regsvr32 SampleProvider.dll” command, to register with the operating system.

On Mac OS, follow these steps:

1. Use Apple Xcode 2.4.1 to open <SDK>/SamplePlugin\build\mac\SampleProvider.Xcodeproj.

2. Build the project.

3. The compiled plug-in named SampleProvider.bundle is located in <SDK>/SamplePlugin\build\mac\build\Debug or <SDK>/SamplePlugin\build\mac\build\release.

4. Follow the steps in “Installing Linguistic Library and Plug-ins” on page 8 and “Exercising Adobe Linguistic Library” on page 9 to install a custom Linguistic Library plug-in and exercise the Linguistic Library.

During your testing, you should notice all words other than “Adobe”, “Lilo” and “misunderstanding” are flagged as misspelled. For hyphenation, “laboratory” may be hyphenated in position 3 or 5 or 2 or 7 (after “b” or the first “r” or first “a” or first “t”); All other words may be hyphenated in the middle. These are due to our hardcoded spelling and hyphenation implementations.

**Build your First Linguistic Library Plug-in Using SampleProvider as a Template**

SampleProvider is the starting point for your first plug-in. To build your first plug-in based on SampleProvider, follow these steps:

1. Copy the entire <SDK>/api and <SDK>/SamplePlugin folder to a new location. If you use a source-code control system to track your code, this could be your initial version.

2. Name the plug-in. You may not want your plug-in to use the same name as SampleProvider. In the following, we assume you name your plug-in MyProvider:
   
   - Open the build/win folder, and rename SampleProvider.vcproj, SampleProvider.sln and SampleProvider.vcxproj, to be MyProvider.vcproj, MyProvider.sln and MyProvider.vcxproj.
• Open MyProvider.vcproj, MyProvider.sln and MyProvider.vcxproj as plain text, and replace all occurrences of “SampleProvider” with “MyProvider”.

• Open the source\win folder, and rename SampleProvider.cpp, SampleProvider.h, and SampleProvider.def to be MyProvider.cpp, MyProvider.h, and MyProvider.def. (This step is not necessary if you did not change the filename included in the vcproj in the previous step).

• Replace all instances of “SampleProvider” in files in the “source” folder, especially the ones in MyProvider.cpp, MyProvider.h, and MyProvider.def.

• Open build/win/info.plist, and replace SampleProvider with MyProvider in the <key>CFBundleIdentifier</key> section.

• Open the build/mac folder, and rename SampleProvider.Xcodeproj to be MyProvider.Xcodeproj. (Since SampleProvider.Xcodeproj is a folder, you just need to rename the folder, to open a Mac OS project on Windows.)

• Open MyProvider.Xcodeproj\project.pbxproj as plain text, and replace all SampleProvider with MyProvider. Alternately, you can open MyProvider.Xcodeproj in Xcode, and change the target filenames, etc.

• Open mac/info.plist, and replace SampleProvider with MyProvider in the <key>CFBundleIdentifier</key> section.

3. Name your provider. Open info.plist and replace LiloSamplePlugin with your provider name in the <key>ALPluginName</key> section.

4. Assign version number to your plug-in. Replace 11.0.21 with your plug-in version under <key>ALPluginVersion</key>.

5. Make your plug-in loadable on Windows:
   • From the command shell, run guidgen to generate a new GUID. Please refer (http://msdn.microsoft.com/en-us/library/kw069h38.aspx) for more info on guidgen utility.
   • Open MyProvider.cpp, and replace the DEFINE_GUID statement with a new GUID value, to define a new CLSID.
   • Find the gRegTable definition, and replace all CLSID strings with the same new GUID value.
     • Open info.plist and replace C5904E75-6564-42CD-94F3-8FD6FF6C7C66 with your GUID in the <key> C5904E75-6564-42CD-94F3-8FD6FF6C7C66 </key>.

NOTE: Do not change the GUID of interface IDs defined in files under the headers folder.

6. Make your plug-in loadable on Mac OS. Linguistic Library uses a different mechanism to load a plug-in on Mac OS. In contrast to Windows, do not change the GUID of CLSID_LiloProviderPlguIn as defined in Server.cpp.

7. In info.plist specify the services under <key>ALPlug-inServices</key>

8. In info.plist specify the languages for each service under <key>ALPlug-inLanguages</key>
9. In info.plist specify the list of capabilities under <key>ALPlug-inCapabilities</key>

10. Change spelling, hyphenation or userdictionary service to your own implementations. You can freely change ALPluginSpellChecker.cpp, ALPluginHyphenation.cpp and ALPluginUserDictionary.cpp, as long as the methods on their corresponding interface are implemented. You can even add new support classes, CPP files, data files, etc. into the project. You also can alter the ALPlug-inProvider implementation as needed.

11. Test your own plug-in. Follow the same steps as for testing SampleProvider. You should see both “LiloSamplePlug-in” and your own provider name as choices for the end user.

**Linguistic Library Plug-in API Reference**

This section provides detailed descriptions of the Linguistics Library plug-in programming interfaces.

**LinguisticPlugin.h**

LinguisticPlugin.h defines commonly used enums and constants.

**Language Code**

The first segment of the file defines the ISO language code. For example, en_US is American English and fr_FR is French. For a detailed list of language codes, see the source code.

**Linguistic Service**

We defined four linguistic services as enums: kLMSpellingService, kLMHyphenationService, kLMThesaurusService, and kLMUserDictionaryService. Adobe Linguistics Library version 6.0 supports only spelling and hyphenation services.

**Property Type**

LM_PluginPropertyType defines the property types that can be set in a plug-in. It is used to determine the capability of a plug-in. Currently, these are used as spell-checker options.

**Hyphenation-point Preference**

There are three types of hyphenation-point preference: kLMHyphenNormalPoint, kLMHyphenPreferredPoint, and kLMHyphenNonpreferredPoint. They are used in Linguistic Library hyphenation implementations to specify how to split a word.
IALPProvider Interface

IALPProvider allows Linguistics Library clients to implement user-interface features. GetPropertyValue() and SetPropertyValue() deal with provider-specific properties (or capabilities/options). The interface is defined as follows:

class IALPProvider : public IUnknown
{
    public:
        virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE
GetPropertyValue(
            /* [in] */ const CHAR *property,
            /* [out] */ ULONG *value) = 0;
        virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE
SetPropertyValue(
            /* [in] */ const CHAR *property,
            /* [in] */ ULONG value) = 0;
};

GetProperty

This function is used by the Linguistic Library to obtain the value of a capability. A value of 0 means false and 1 means true. Syntax:

HRESULT STDMETHODCALLTYPE GetPropertyValue (const CHAR *property,
    ULONG *value);

**Table 1 GetProperty Parameter Table**

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property</td>
<td>[in]</td>
<td>The capability of the value requested.</td>
</tr>
<tr>
<td>value</td>
<td>[out]</td>
<td>The value of the capability.</td>
</tr>
</tbody>
</table>

SetProperty

This function is used by the Linguistic Library to set the value of the given capability. A value of 0 means false and 1 means true. Syntax:

HRESULT STDMETHODCALLTYPE SetPropertyValue (const CHAR *property,
    ULONG *value);

**Table 2 SetProperty Parameter Table**

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property</td>
<td>[in]</td>
<td>The capability whose value will be set.</td>
</tr>
</tbody>
</table>
| value   | [out]| The value to set for the capability.
IALPSpellChecker interface

IALPSpellChecker provides spelling services:
- IsWord() checks the spelling of a word.
- GetSuggestionList() returns an array of suggestions for a word.

class IALPWordBasedSpellChecker : public IUnknown
{
public:

    virtual /*[helpstring]*/ HRESULT STDMETHODCALLTYPE Initialize(
        /* [in] */   const CHAR* inLanguage) = 0;

    virtual /*[helpstring]*/ HRESULT STDMETHODCALLTYPE SetContext(
        /* [in] */   ULONG context
        // LM_SentencePosition ) = 0;

    virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE IsWord(
        /* [in] */   const OLECHAR* word,
        /* [in] */   ULONG length,
        /* [retval][out] */ ULONG *outFailedReason
        // LM_SpellingFailedReason) = 0;

    virtual /*[helpstring]*/ HRESULT STDMETHODCALLTYPE
    GetSuggestionList(
        /* [in] */   const OLECHAR* word,
        /* [in] */   ULONG length,
        /*[out,retval]*/ IEnumALPString **outSuggestionList)=0;
};

IsWord

This function is used by Linguistic Library to verify the spelling of a word. It takes a word and returns true if the word is spelled correctly; otherwise, false. Syntax:

HRESULT STDRESULTDMETHODCALLTYPE IsWord (OLECHAR word, ULONG length, ULONG *outFailedReason);

TABLE 4 IsWord Parameter Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>[in]</td>
<td>The input word</td>
</tr>
<tr>
<td>length</td>
<td>[in]</td>
<td>The input word length</td>
</tr>
<tr>
<td>outFailedReason</td>
<td>[out]</td>
<td>The failing reason of the word</td>
</tr>
</tbody>
</table>
**GetSuggestionList**

This function is used by Linguistic Library to get a list of suggestions for a misspelled word. It takes a word and returns a list of suggestions. Syntax:

```c
HRESULT STDMETHODCALLTYPE GetSuggestionList (OLECHAR* word, ULONG length, IEnumALPString **outSuggestionList);
```

**TABLE 4 CorrectWord Parameter Table**

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>[in]</td>
<td>The input word</td>
</tr>
<tr>
<td>length</td>
<td>[in]</td>
<td>The input word length</td>
</tr>
<tr>
<td>outSuggestionList</td>
<td>[out]</td>
<td>An array of suggestions.</td>
</tr>
</tbody>
</table>

**IALPHyphenation Interface**

ILiloHyphenation provides three functions for hyphenation services:
- CutWord() divides a word.
- GetHyphenationPoints() returns hyphenation points.

```c
class IALPHyphenation : public IUnknown
{
public:

    virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE InitHyphenation(
        /* [in] */   const CHAR *inLanguage) = 0;

    virtual/*[helpstring]*/HRESULT STDMETHODCALLTYPE GetHyphenationPoints(
        /* [in] */   const OLECHAR* word,
        /* [in] */   ULONG length,
        /* [out] */IEnumHyphenationPoint **outEnumPoints) = 0;

    virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE CutWord(
        /* [in] */   const OLECHAR* word,
        /* [in] */   ULONG length,
        /* [in] */ALHyphenationPoint* point,
        /* [out] */   BSTR* outLeftPart,
        /* [out] */   BSTR* outRightPart) = 0;
};
```
GetHyphenationPoints

This function returns a list of hyphenation points and corresponding preference values for a given word. Syntax:

```c
HRESULT STDMETHODCALLTYPE GetHyphenationPoints (OLECHAR* word, ULONG length, IEnumHyphenationPoint **outEnumPoints);
```

**Table 5 GetHyphenationPoints Parameter Table**

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>[in]</td>
<td>The word to be hyphenated</td>
</tr>
<tr>
<td>length</td>
<td>[in]</td>
<td>The length of the word to be hyphenated</td>
</tr>
<tr>
<td>outEnumPoints</td>
<td>[out]</td>
<td>The number of hyphenation points.</td>
</tr>
</tbody>
</table>

CutWord

This function is used by Linguistic Library to obtain the left and right parts of a hyphenated word. It is necessary to have this function as in some languages; the spelling of a word is changed after it is hyphenated. Syntax:

```c
HRESULT STDMETHODCALLTYPE CutWord (OLECHAR* word, ULONG length, ALHyphenationPoint* point, BSTR* outLeftPart, BSTR* outRightPart);
```

**Table 6 CutWord Parameter Table**

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>[in]</td>
<td>The word to be broken at a given hyphenation point.</td>
</tr>
<tr>
<td>length</td>
<td>[in]</td>
<td>The length of the word.</td>
</tr>
<tr>
<td>point</td>
<td>[in]</td>
<td>The index point where the word needs be separated.</td>
</tr>
<tr>
<td>outLeftPart</td>
<td>[out]</td>
<td>The left part of the hyphenated word.</td>
</tr>
<tr>
<td>outRightPart</td>
<td>[out]</td>
<td>The right part of the hyphenated word.</td>
</tr>
</tbody>
</table>

IALPUserDictionary Interface

IALPUserDictionary provides the following functions:
- AddWord/Addwords to add a single or multiple words in the user dictionary.
- AddWordPair to add a misspelled and Correct word pair.
- EraseWord/EraseAllWords to erase a single word or all the words from the dictionary.

```c
class IALPUserDictionary : public IUnknown
{
    public:

        // for spelling service
```
virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE Initialize(
    /* [in] */ const CHAR *inLanguage,
    /* [in] */ const CHAR *inFilePath) = 0;

virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE AddWord(
    /* [in] */ BSTR inWord) = 0;

virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE AddWords(
    /* [in] */ SHORT inCount,
    /* [in] */ BSTR *inWordArray) = 0;

virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE EraseWord(
    /* [in] */ BSTR inWord) = 0;

virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE AddWordPair(
    /* [in] */ BSTR inKey,
    /* [in] */ BSTR inData) = 0;

virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE EraseAllWords() = 0;


AddWord/AddWords

The function AddWord adds a single word in the user dictionary and the function AddWords adds a list of words in the user dictionary. Syntax:

HRESULT STDMETHODCALLTYPE AddWord( BSTR inWord) ;
HRESULT STDMETHODCALLTYPE AddWords(SHORT inCount,
    BSTR *inWordArray);

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inWord</td>
<td>[in]</td>
<td>The word to be added.</td>
</tr>
<tr>
<td>inCount</td>
<td>[in]</td>
<td>The number of words to be added.</td>
</tr>
<tr>
<td>inWordArray</td>
<td>[in]</td>
<td>The list of words to be added.</td>
</tr>
</tbody>
</table>

EraseWord/EraseAllWords

The function EraseWord erases a single word from the user dictionary and the function EraseAllWords deletes the entire user dictionary. Syntax:
HRESULT STDMETHODCALLTYPE EraseWord ( BSTR inWord) ;
HRESULT STDMETHODCALLTYPE EraseAllWords();

**Table 7 EraseWord Parameter Table**

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inWord</td>
<td>[in]</td>
<td>The word to be deleted.</td>
</tr>
</tbody>
</table>

**AddWordPair**

The function AddWordPair adds a misspelled and correct word pair to the user Dictionary. Whenever the user enters the misspelled word the first suggestion shown is the correct word from the word pair added in the user dictionary.  
Syntax:

HRESULT STDMETHODCALLTYPE AddWordPair(BSTR inKey, BSTR inData);

**Table 8 AddWordPair Parameter Table**

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inKey</td>
<td>[in]</td>
<td>The key to be added.</td>
</tr>
<tr>
<td>inData</td>
<td>[in]</td>
<td>The data to be added.</td>
</tr>
</tbody>
</table>