

**ADOBE® ILLUSTRATOR® CS3**

# **ADOBE ILLUSTRATOR CS3 PROGRAMMER'S GUIDE**



© 2007 Adobe Systems Incorporated. All rights reserved.

*Adobe Illustrator CS3 Programmer's Guide*

Adobe, the Adobe logo, Illustrator, PageMaker, Photoshop, and FrameMaker are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Apple and Mac OS are trademarks of Apple Computer, Inc., registered in the United States and other countries. All other trademarks are the property of their respective owners.

The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in this document. The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.

# Contents

	<b>Preface .....</b>	<b>5</b>
	Terminology .....	5
	Notational conventions .....	5
	Supporting documentation .....	5
	Sample code .....	5
<b>1</b>	<b>Overview .....</b>	<b>6</b>
	What is a plug-in? .....	6
	A brief history of the Adobe Illustrator API .....	6
	Anatomy of a plug-in .....	7
	Types of plug-ins .....	8
	Action plug-ins .....	9
	Plug-in file formats .....	9
	Plug-in filters .....	9
	Plug-in menu commands .....	9
	Plug-in notifiers and timers .....	9
	Plug-in-group plug-ins .....	10
	Plug-in tools .....	10
	Combining multiple plug-in types .....	10
	Where plug-ins live .....	10
	What defines a plug-in? .....	11
	PiPL resources .....	11
	Plug-in management .....	11
	Plug-in entry point and messages .....	11
	Message actions: callers and selectors .....	12
	Core message actions .....	12
	Reload and unload messages .....	13
	Start-up and shut-down messages .....	13
	About messages .....	13
	Notifiers .....	14
	Handling callers and selectors .....	14
	Message data .....	14
	Illustrator API .....	15
	Suites .....	15
	Acquiring and releasing suites .....	16
	Publishing suites .....	17
	Binary compatibility .....	17
	Illustrator's artwork as seen by a plug-in .....	18
	Cross-platform support .....	18
	Memory .....	19
	Resources .....	19

	Byte information and structures .....	19
	System requirements .....	19
	Helper classes .....	19
	Plug-in adapters .....	20
	About Plug-ins menu .....	20
	Next steps .....	20
<b>2</b>	<b>Tutorial .....</b>	<b>22</b>
	PiPL resource and plug-in entry point .....	22
	Handling callers and selectors .....	22
	Using suites and callback functions .....	22
	Acquiring and releasing suites and calling suite functions .....	23
	Illustrator API suites .....	23
	Callers and selectors .....	23
	Caller: kSPIInterfaceCaller, selector: kSPIInterfaceStartupSelector .....	23
	Caller: kSPIInterfaceCaller, selector: kSPIInterfaceShutdownSelector .....	25
	Caller: kSPAccessCaller, selector: kSPAccessReloadSelector .....	25
	Caller: kSPAccessCaller, selector: kSPAccessUnloadSelector .....	26
	Plug-in type-specific messages .....	26
	Filter plug-ins .....	26
	Tool plug-ins .....	27
	Action plug-ins .....	28
	Menu Plug-ins .....	30
	Next steps .....	30
<b>3</b>	<b>Plug-in Property Lists .....</b>	<b>31</b>
	PiPL samples .....	31
	PiPL structure .....	31
	Required PiPL properties .....	32
	Optional PiPL properties .....	32
	Export property .....	33
	For more information .....	34
	<b>Index .....</b>	<b>35</b>

# Preface

This document introduces the Adobe® Illustrator® API for plug-ins.

- [Chapter 1, “Overview”](#) describes the basic concepts behind Illustrator plug-in programming.
- [Chapter 2, “Tutorial”](#) describes plug-in programming fundamentals using an example project.
- [Chapter 3, “Plug-in Property Lists”](#) describes PiPL resources in detail.

## Terminology

**API** — Application programming interface.

**Application** — Illustrator CS3, unless otherwise specified.

**PiPL** — Plug-in property list.

**SDK** — Software development kit for the application.

## Notational conventions

- SDK root folder — `<SDK>` indicates your locally installed SDK root folder. The actual root location depends on the installation and operating system.
- Menu names — The right angle bracket, `>`, is used to indicate menu hierarchies. For example, `File > Open` refers to the Open menu item on the File menu.
- Computer input or output (including source code or API types) is shown in monospace font.

## Supporting documentation

The following documents are companions to this guide:

- *Adobe Dialog Manager CS3 Programmer's Guide* — Describes concepts involved in programming the cross-platform user-interface toolkit used in Illustrator. See `<SDK>/docs/guides/adm-guide.pdf`.
- *Adobe Illustrator CS3 Porting Guide* — Describes platforms that are supported for plug-in development and issues related to porting plug-ins across different versions of the Illustrator API. See `<SDK>/docs/guides/porting-guide.pdf`.
- *Adobe Illustrator API Reference* — Describes the suites and functions in the Illustrator API in detail. See the `<SDK>/docs/references/` folder.

## Sample code

Sample plug-ins are provided in the sample code folder in the SDK. One sample is the Tutorial project. [Chapter 2, “Tutorial,”](#) describes plug-in programming fundamentals using Tutorial as a sample project.

# 1 Overview

This chapter introduces the Adobe® Illustrator® plug-in architecture. It shows how plug-ins interact with Illustrator and gives an idea of the power of plug-ins.

## What is a plug-in?

A plug-in is a library that extends or changes the behavior of Illustrator. On Windows, a plug-in is built as a DLL (dynamic-link library). On Macintosh, a plug-in is built as a bundle that contains a shared library.

The plug-in architecture in Illustrator is very powerful. Plug-ins have access to Illustrator's elegant vector, text, and raster engines, as well as Illustrator's user interface. The API is a fundamental part of the application; in fact, most of Illustrator itself is implemented as plug-ins.

You can create plug-ins that add new tools to the drawing, shading, and raster tools already in Illustrator. Plug-ins can turn Illustrator into a powerful CAD, cartographic, or other custom design application. Users can add or remove plug-ins to quickly and easily customize Illustrator to their needs.

The Illustrator API offers several benefits to plug-in developers. Because Illustrator handles large application tasks like printing and saving files, you can concentrate on the implementation of your plug-in's unique features. Plug-ins do not need to support undo or window updating; these are handled by the API and are invisible to the plug-in. This translates into shorter development cycles.

Your plug-in also can take advantage of Adobe Dialog Manager (ADM), a cross-platform, user-interface API. Plug-ins that use ADM floating panels are completely integrated with Illustrator; the panels can be docked, undocked, and controlled as if they were built into Illustrator.

## A brief history of the Adobe Illustrator API

The Illustrator API first appeared in version 5.0. It supported one type of plug-in, filters. This was extended in Adobe Illustrator 5.5 to include file formats. The 5.x APIs displayed characteristics of many early API design efforts: the interface was monolithic, incorporating enough function to achieve its intended purpose, but not allowing for future expansion. A single callback function table was provided, with no means to extend or update it. Platform abstraction was minimal, and interaction with the user was restricted to modal.

The Illustrator 6.0 API began addressing these limitations. A modular and extensible approach was used to export the application's functionality. Callback functions were organized into *suites* that could be easily replaced or extended. The plug-in types were abstracted and extended to include tools, floating windows, menu items, events, and combinations of these types. The first steps towards platform abstraction were taken.

The Illustrator 7.0 API refined prior efforts. The API was extended to be truly cross-platform (the Windows version of Adobe Illustrator jumped from version 4.2 directly to 7.0), including a complete set of user-interface suites. The plug-in management core was generalized for not only cross-platform, but also cross-product use. More of Illustrator's core functionality was implemented through plug-ins, allowing the application's behavior to change without modifying the application itself.

The most significant change in the Illustrator 8.0 API was the switch from fixed numbers to floating-point numbers. All suites that used fixed parameters were updated to use floating-point parameters; in some cases, Adobe changed function names to reflect this. The Illustrator 8.0 API also included more than a dozen new suites, many corresponding to new user features like plug-in groups, cursor snapping, and actions.

Illustrator 9.0 on the Macintosh stopped loading plug-ins containing Motorola 68K code. While early plug-in APIs from Adobe would run 68K plug-ins from a PowerPC application or vice-versa, this mechanism was not provided to Illustrator 9 plug-ins. Only plug-ins with the Illustrator 9 version information in the PiPL were recognized and included in Illustrator 9's initial start-up process; however, Illustrator 6 and 5.5 plug-ins were supported by adapters.

The Illustrator 10.0 API changed API structures on the Macintosh to use PowerPC alignment (four-byte aligned). Earlier APIs used 68K alignment (two-byte aligned). Illustrator plug-ins for the Macintosh had to be rebuilt with the Illustrator 10 SDK, to be compatible with Illustrator 10. Illustrator plug-ins for Windows built with the Illustrator 9 SDK or earlier were not affected and remained compatible with Illustrator 10.

**NOTE:** Originally, Illustrator was a 68K application. When Apple moved to the PowerPC platform, the Illustrator API maintained 68K alignment for backward compatibility. This caused a small performance loss, so the alignment was changed; this change broke backward compatibility.

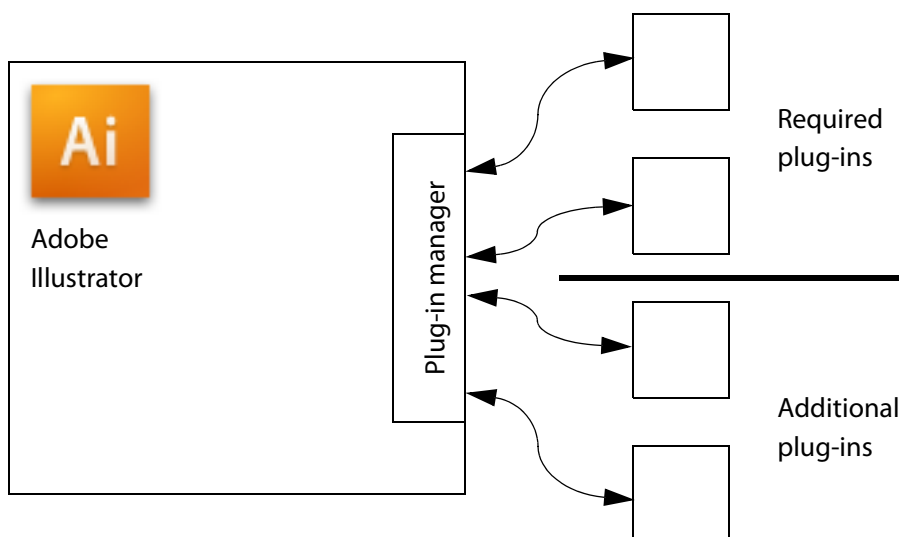
Illustrator CS1 (Illustrator version 11.0) integrated a new text engine, the Adobe Text Engine (ATE). The text API was revised completely. The text API from Illustrator 10.0 and earlier was obsoleted (`AIText`, `AITextFaceStyle`, `AITextPath`, `AITextRun`, `AITextStream`, `AITextStyle`, etc.). The new ATE API provided increased functionality and control over typography to Illustrator plug-ins (see `IText` and `ATESuites`). This change broke backward compatibility. Illustrator plug-ins that used the obsolete text API had to be rebuilt with the Illustrator 11 SDK and ported to use the new ATE API.

The Illustrator CS2 API (Illustrator version 12.0) introduced Unicode support. Unicode encoding should be used for any strings displayed in the user interface. On the Macintosh, the object-file format for Illustrator plug-ins changed to Mach-O. The old format was PEF, a preferred executable format. Moving forward, Mach-O is Apple's preferred object-file format for the Mac OS X platform.

The Illustrator CS3 API (Illustrator version 13.0) introduced support for universal binary plug-ins on the Macintosh. Universal binaries run on Macintosh computers using PowerPC or Intel microprocessors and deliver optimal performance for both architectures in one package. The Apple Xcode development tool is the most convenient way to produce universal binaries; therefore, Adobe switched to Xcode from CodeWarrior for the development of Illustrator CS3 plug-ins. Documentation on this change and other API changes is in *Adobe Illustrator CS3 Porting Guide*.

## Anatomy of a plug-in

Like most programs, Illustrator plug-ins contain both code and data. The *Illustrator plug-in manager* loads and executes a plug-in's code when required, sending various messages to the plug-in. The plug-in manager also unloads plug-ins that are no longer needed. See the following figure.



Plug-ins are notified by Illustrator when they have just been loaded or are about to be unloaded, permitting them to restore or save any state information.

## Types of plug-ins

This section describes the different types of Illustrator plug-ins you can create. A single plug-in file can contain multiple plug-in types. For example, a shape creation plug-in may implement several plug-in filters and a plug-in tool. Plug-in types are listed in the following table and described more fully after the table.

Plug-in Type	What it does
Action	Playback or register actions.
Effects	Add menu items to the Effects menu.
File format	Add file types to the Open, Save, and Export commands.
Filter	Add menu items to the Filter menu.
Menu command	Add menu items to the general menu structure.
Notifier	Receive and process art-state events.
Plugin group	Maintain “display” art that is associated with another art object
Suite	Implement and publish your own suite of callback functions.
Timer	Receive periodic notification.
Tool	Add tools to the Tools panel.
Transform again	Set the transform-again feature.



## Action plug-ins

Action plug-ins are used to execute Illustrator commands. An action plug-in can register itself so it can be recordable via the Actions panel. For more information, see `AIActionManagerSuite` in *Adobe Illustrator API Reference*.

## Plug-in file formats

Plug-in file formats are used to extend the number of file types Illustrator can read and write. Plug-ins indicate which file formats they support during initialization, specifying the supported names and file types (or extensions). One plug-in can register as many formats as desired.

The file types supported by a file-format plug-in can appear in Illustrator's Export, Save, and Open dialogs, depending on the options specified when the new file type is added.

For more information, see `AIFileFormatSuite` in *Adobe Illustrator API Reference*.

## Plug-in filters

Plug-in filters appear under the Filter menu and are used to create or manipulate Illustrator artwork. Typically, filters present a modal interface to the user, who can set parameters before executing.

**NOTE:** In other applications (including Adobe PageMaker® and Adobe FrameMaker®), the term “filter” or “filter plug-in” sometimes is used to describe software that reads and writes non-native files (e.g., TIFF or JPEG files). In Illustrator, these are called file-format plug-ins. Illustrator uses the term “filter plug-in” in a way similar to Adobe Photoshop: a filter plug-in modifies the artwork in an algorithmic fashion.

Illustrator updates the Repeat and Undo menus automatically, making filters one of the simplest plug-in types to create.

For more information, see `AIFilterSuite` in *Adobe Illustrator API Reference*.

## Plug-in menu commands

Plug-in menus are used to add menu items to Illustrator's menu structure other than the Filter menu. A typical use of this plug-in type is to add a Hide/Show Window menu item to Illustrator's Window menu.

Plug-in menu commands can be added at several places in the menu structure.

For more information, see `AIMenuSuite` in *Adobe Illustrator API Reference*.

## Plug-in notifiers and timers

Plug-in notifiers and timers are used by a plug-in to have Illustrator inform it of certain events.

A *notifier plug-in* is notified when the state of an Illustrator document changes. For example, a plug-in may request to be notified when the selection state changes. A notifier plug-in registers for one or more notifications during start-up.

A *timer plug-in* is notified at regular time intervals. For example, a timer plug-in may request to be notified once a second.

For more information, see `AINotifierSuite` and `AITimerSuite` in *Adobe Illustrator API Reference*.

## Plugin-group plug-ins

Plugin-group plug-ins maintain one or more plug-in groups. A plug-in group is a special art object that contains editable art as well as art that is displayed but not editable. A plugin-group plug-in is responsible for regenerating the display art (or result art) whenever there is a change in the edit art. Plug-in groups are used to make special art types like Live Blends and Brushes.

For more information, see `AIPluginGroupSuite` in *Adobe Illustrator API Reference*.

## Plug-in tools

Plug-in tools add an icon to the Tools panel, expanding the number of tools available to the user. Many standard Illustrator tools, including the knife tool, shape tools, and twirl tool, are implemented as plug-in tools.

When selected, a tool plug-in can track the mouse, determine which artwork was selected, and act on it. For example, a tool might create or distort objects. Some things are handled automatically for plug-in tools, like scrolling the window.

For more information, see `AIToolSuite` in *Adobe Illustrator API Reference*.

## Combining multiple plug-in types

As mentioned before, it is likely one plug-in file implements multiple plug-in types. A plug-in also may need to add multiple instances of a single plug-in type. The plug-in API supports both these cases.

## Where plug-ins live

Illustrator's Plug-ins folder is in the following locations:

Windows: `C:\Program Files\Adobe\Adobe Illustrator CS3\Plug-ins\`

Macintosh: `/Applications/Adobe Illustrator CS3/Plug-ins/`

A user-specific Plug-ins folder, which may be appropriate to use in multi-user systems, is located as follows:

Windows XP: `C:\Documents and Settings\<username>\Application Data\Adobe\Adobe Illustrator CS3 Settings\Plug-ins\`

Windows Vista: `C:\Users\<username>\AppData\Roaming\Adobe\Adobe Illustrator CS3 Settings\Plug-ins\`

Macintosh `/Users/{username}/Library/Application Support/Adobe/Adobe Illustrator CS3/Plug-ins`

Optionally, an additional folder can be specified using Illustrator's Additional Plug-ins Folder preference.

In general, each plug-in type is in a specific subfolder; for example, tool plug-ins are in a folder named `Tools`.

## What defines a plug-in?

On Windows, an Illustrator plug-in is a DLL (dynamic-link library). On Macintosh, an Illustrator plug-in is a bundle that contains a shared library. An Illustrator plug-in has the following characteristics:

- A file extension of `.aip`; for example, `CoolEffect.aip`.
- A valid plug-in PiPL resource. The PiPL resource contains information about your plug-in. Illustrator considers only those files with PiPL resources to be potential plug-ins. Files with the correct properties are added to the plug-in list.
- A code entry point containing binary code that can run on the target platform. The entry point is specified in the PiPL resource and is called with several messages telling it which actions to take.

## PiPL resources

A plug-in property list (PiPL) resource contains properties used by the Illustrator plug-in manager, including the following:

- The type of the plug-in, given by the `kind` property.
- The calling mechanism for the plug-in code, given by the `ivrs` property.
- The entry point of the plug-in, given by a `code descriptor` property.

Illustrator considers only files with a valid PiPL to be potential plug-ins. PiPL properties are defined using native platform resources. For more information about PiPL resources and samples, see [Chapter 3, “Plug-in Property Lists”](#).

## Plug-in management

When Illustrator is launched, only plug-ins with a valid PiPL and code entry point are recognized and included in the initial start-up process. Each plug-in is loaded into and unloaded from memory as needed by Illustrator. A plug-in needs to be written assuming it is not always in memory. This is why a plug-in should save and restore state information during unload and reload. A plug-in can expect certain services from the application. Because a plug-in may be unloaded, Illustrator provides a means of storing important data when the plug-in is unloaded. Each time a plug-in is called, it is given enough information to accomplish the action to be performed.

The loading order of plug-ins becomes important when one plug-in depends on a resource provided by another, as the resource providing plug-in must be loaded first. As mentioned above, plug-ins that export one or more suites must declare (in the PiPL resource) what they export. Illustrator uses this information when loading and executing plug-ins, ensuring that suites and other resources are available.

## Plug-in entry point and messages

The Illustrator plug-in manager communicates with your plug-in by loading the plug-in code into memory if necessary, then calling the entry point given by the code-descriptor property in the PiPL. By convention, the entry point is called `PluginMain` and is compiled with C linkage:

```
extern "C" ASAPI AErr PluginMain(char* caller, char* selector, void* message);
```

Three arguments are passed to the `PluginMain` function; collectively, they make up a *message*.

The first two parameters represent the *message action*, describing what the plug-in is supposed to do, as described in the following section. The third parameter is a pointer to a data structure, which varies depending on the message action. When you determine the message action, you typecast the data in the `message` parameter as needed.

The result of the function is an error code.

## Message actions: callers and selectors

Each time your plug-in is called, it receives a message action from Illustrator. The message action notifies your plug-in that an event happened or tells your plug-in to perform an action.

The message action passed to your plug-in consists of two identifiers:

- The *caller* identifies the sender of the message (PICA, the host application, or a plug-in) and a general category of action.
- The *selector* specifies the action to take within the category of action. All plug-ins receive at least four message actions: reload, unload, startup and shutdown. In addition, your plug-in may receive additional message actions specific to the plug-in type.

For example, Illustrator sends a plug-in a menu a message action based on these two strings, when a menu item added by the plug-in is clicked:

```
// Caller and Selector from the Menu Suite
#define kCallerAIMenu          "AI Menu"
#define kSelectorAIGoMenuItem "AI Go"
```

The caller and selector identifiers are C strings. By convention, each caller string has a prefix. This is so new message actions can be easily defined by other applications and plug-ins, with little chance of conflict. For example, callers and selectors from Illustrator suites use the prefix "AI", while those from PICA use the prefix "SP".

Illustrator message actions are used to indicate events in which a plug-in has interest. Information on the callers and selectors supported by the API is given by the Plug-in Callers and Plug-in Selectors pages in *Adobe Illustrator API Reference*.

## Core message actions

The following table contains the set of core message actions received by all plug-ins and corresponding actions your plug-in should take.

Caller	Selector	Action to perform
kSPAccessCaller ("SP Access")	kSPAccessReloadSelector ("Reload")	Restore any state information (globals).
	kSPAccessUnloadSelector ("Unload")	Save any state information (globals).

Caller	Selector	Action to perform
kSPInterfaceCaller ("SP Interface")	kSPInterfaceStartupSelector ("Startup")	Initialize globals and add features to the application.
	kSPInterfaceShutdownSelector ("Shutdown")	Free globals, remove features from the application, and save preferences.

## Reload and unload messages

Whenever a plug-in is loaded into memory or unloaded from memory, Illustrator sends it an *access* message action:

```
#define kSPAccessCaller          "SP Access"
#define kSPAccessUnloadSelector "Unload"
#define kSPAccessReloadSelector "Reload"
```

The message action contains the *access caller* and a *reload* or *unload* selector. This is your plug-in's opportunity to set up, restore, or save state information. The access caller/selectors bracket all other callers and selectors.

Access messages bracket all other messages. Reload is the first message your plug-in receives; Unload, the last. At these times, your plug-in should not acquire or release suites other than those built into Illustrator.

## Start-up and shut-down messages

Illustrator has two core interface message actions, where the plug-in can interact with the application:

```
#define kSPInterfaceCaller      "SP Interface"
#define kSPInterfaceStartupSelector "Startup"
#define kSPInterfaceShutdownSelector "Shutdown"
```

When Illustrator is launched, it sends a "startup" message to each plug-in it finds. This allows your plug-in to allocate global memory, add user-interface items to Illustrator, register suites, or perform other initialization. The start-up message action consists of the interface caller (`kSPInterfaceCaller`) and start-up selector (`kSPInterfaceStartupSelector`).

When the user quits Illustrator, it sends each plug-in a "shutdown" message. The start-up message action consists of the interface caller (`kSPInterfaceCaller`) and start-up selector (`kSPInterfaceShutdownSelector`). Shut-down is intended for flushing files and preserving preferences, not destruction. A plug-in that exports a suite should not dispose of its plug-in globals or suite information, since it may be called after its own shut-down by another plug-in's shut-down. For example, if your plug-in implements a preferences suite that other plug-ins use, they may call you in their shut-down handlers after you already shut down.

## About messages

When a plug-in receives the "About" message action (`kSPInterfaceAboutSelector`) it displays a dialog with information about the plug-in.

**NOTE:** As of Illustrator CS3 the `kSPInterfaceAboutSelector` selector is not sent anymore. Plug-ins no longer need to respond to this message. To display company contact and copyright information, follow the guidelines in ["About Plug-ins menu" on page 20](#).

## Notifiers

Some message actions also are referred to as *notifiers*, indicating something in Illustrator was changed by the user; for example, when the user selects an object.

Plug-ins must register for the notifiers in which they are interested. The Notifier suite is used to register and remove notification requests (see `AINotifierSuite`).

Plug-ins also can create their own notifiers, which can be used to broadcast changes to other plug-ins.

## Handling callers and selectors

Your plug-in's organization is based largely on the messages it receives. The main routine of your plug-in must first determine the message action, using the caller and selector parameters. For example:

```
extern "C" ASAPI ASErr PluginMain(char* caller, char* selector, void* message)
{
    ASErr error = kNoErr;
    if ( strcmp( caller, kSPAccessCaller ) == 0 ) {
        // Handle Reload and Unload
        if ( strcmp( selector, kSPAccessReloadSelector ) == 0 )
            error = MyRestoreGlobals( message );
        else if ( strcmp( selector, kSPAccessUnloadSelector ) == 0 )
            error = MySaveGlobals( message );
    } else if ( strcmp( caller, kSPIInterfaceCaller ) == 0 ) {
        // Handle Startup and Shutdown
        if ( strcmp( selector, kSPIInterfaceStartupSelector ) == 0 )
            error = MyStartupPlugin( message );
        else if ( strcmp( selector, kSPIInterfaceShutdownSelector ) == 0 )
            error = MyShutdownPlugin( message );
    } else if ( strcmp( caller, kCallerAIMenu ) == 0 &&
        strcmp( selector, kSelectorAIGoMenuItem ) == 0 ) {
        // Handle menu message
        error = MyHandleMenu( message );
    }
    return error;
}
```

## Message data

The last argument passed to your plug-in entry point is a pointer to a message data structure, which contains information appropriate to the message action. For example, when a mouse-clicked message action is received, the message data structure contains the mouse position.

The contents of the message data structure depend on the message action and are not completely known until your plug-in identifies this. While the contents of the message data vary, by convention all message data structures begin with the common fields that are grouped into the `SPMessageData` structure:

```
typedef struct SPMessageData {
    long SPCheck;
    struct SPPlugin *self;
    void *globals;
    struct SPBasicSuite *basic;
} SPMessageData;
```

If this is a valid message, the `SPCheck` field contains `kSPValidSPMessageData`.

The `self` field is a reference to the plug-in being called. The reference to the running plug-in's `self` is used to add plug-in suites, adapters, and other plug-in data to Illustrator. Illustrator stores this value with the added data. It is used to recall your plug-in as needed.

The `globals` pointer is for use by your plug-in, to preserve any information between calls that it needs. Usually, it is a pointer to a block of memory allocated by your plug-in at start-up. This value is preserved by Illustrator when your plug-in is unloaded and passed back to the plug-in each time it is called. Plug-ins use this block to store any state information they need to maintain between unload and reload.

**NOTE:** It is important that the memory for globals be allocated using Illustrator's memory-allocation APIs; otherwise, the memory may be destroyed by the operating system when a plug-in is unloaded.

The `basic` field is a pointer to the Basic suite (see `SPBasicSuite`), which allows your plug-in to acquire other suites and provides basic memory management. See [“Suites” on page 15](#).

When Illustrator or a plug-in wants to send a message to your plug-in, it passes in a relevant message data structure. Some examples are given below.

Caller	Message type	Description
<code>kSPAccessCaller</code>	<code>SPAccessMessage</code>	Contains <code>SPMessageData</code> .
<code>kSPInterfaceCaller</code>	<code>SPInterfaceMessage</code>	Contains <code>SPMessageData</code> .
<code>kCallerAIMenu</code>	<code>AIMenuMessage</code>	Contains <code>SPMessageData</code> and a reference to a menu item that was chosen.

Once a plug-in identifies the message action via the caller and selector parameters, it casts the message parameter to access further message data. For example:

```
if (strcmp(caller, kSPAccessCaller) == 0) {
    SPAccessMessage* accessMsg = static_cast<SPAccessMessage*>(message);
    // access accessMsg
}
else if (strcmp(caller, kSPInterfaceCaller) == 0) {
    SPInterfaceMessage* interfaceMsg = static_cast<SPInterfaceMessage*>(message);
    // access interfaceMsg
}
else if (strcmp(caller, kCallerAIMenu) == 0) {
    AIMenuMessage* menuMsg = static_cast<AIMenuMessage*>(message);
    // access menuMsg
}
```

## Illustrator API

### Suites

The Illustrator plug-in manager calls a plug-in through the plug-in's entry point, sending various messages as described in the previous section. When a plug-in is active, it needs a way to perform actions within Illustrator. The mechanism for this is *plug-in suites*, which are one or more related functions grouped together in a C structure.

Functions are grouped into suites based on the services they provide; for example the Path Suite (see `AIPathSuite`) contains functions that create and manipulate paths and segments. For detailed documentation on the suites and functions provided, see *Adobe Illustrator API Reference*.

Illustrator's suite architecture, known as the *Plug-in Component Architecture (PICA)*, also is found in the latest versions of Adobe Photoshop and other Adobe applications. The former term for PICA was Suite Pea (SP). Suites that are part of PICA all start with the suffix SP; for example, `SPBasicSuite`.

Suites fall into two general categories: those that implement a plug-in type (see [“Types of plug-ins” on page 8](#)) and those that provide general functions. Suites that provide general functions make up most of the API; they provide a wide range of capabilities for manipulating text, gradients or raster images, or performing math functions. The following table lists several major suites and what they do.

Suite Name	Suite	Services provided
Art suite	<code>AIArtSuite</code>	Access the artwork tree.
Block suite	<code>AIBlockSuite</code> <code>SPBlocksSuite</code>	Allocate and free memory.
Custom Color suite	<code>AICustomColorSuite</code>	Create or work with custom colors.
Document list suite	<code>AIDocumentListSuite</code>	Create or work with documents
Document suite	<code>AIDocumentSuite</code>	Get and set document information.
Gradient suite	<code>AIGradientSuite</code>	Create or work with gradients.
Group suite	<code>AIGroupSuite</code>	Make clipped groups.
Layer suite	<code>AILayerSuite</code>	Get information about layers.
Path suite	<code>AIPathSuite</code>	Work with Illustrator paths.
Random suite	<code>AIRandomSuite</code>	Generate random numbers.
Raster suite	<code>AIRasterSuite</code>	Work with raster objects.
Real Math suite	<code>AIRealMathSuite</code>	Many useful math functions.
Tag suite	<code>AITagSuite</code>	Associate information with art objects.
Text frame suite	<code>AITextFrameSuite</code>	Work with text objects.

For detailed descriptions of suites and their associated functions, see *Adobe Illustrator API Reference*.

## Acquiring and releasing suites

Before you can use a function in a suite, you must first *acquire* the suite. When the suite's functions are no longer needed, your plug-in must *release* the suite.

It is important to release suites so the Illustrator plug-in manager can run optimally. The plug-in manager uses the acquire/release mechanism to determine when plug-ins can be unloaded to free memory.

When your plug-in is first called, it “knows about” only the Basic suite (see `SPBasicSuite`), which was introduced earlier in this chapter (as part of the `SPMessageData` structure). The Basic suite is used to acquire and release other suites.



The following code snippet shows how to pop an alert on start-up using the ADM Basic suite (see `ADMBasicSuite`).

```
extern "C" ASAPI ASErr PluginMain(char* caller, char* selector, void* message)
{
    ASErr error = kASNoError;
    if (strcmp(caller, kSPIInterfaceCaller)==0) {
        if (strcmp(selector, kSPIInterfaceStartupSelector)==0) {
            SPMessagesData* messageData = (SPMessagesData*)message;
            SPBasicSuite* sSPBasic = messageData->basic;
            ADMBasicSuite10* sADMBasic = nil;
            error = sSPBasic->AcquireSuite(kADMBasicSuite, kADMBasicSuiteVersion10,
                (const void**)&sADMBasic);
            if (error == kNoError) {
                sADMBasic->MessageAlert("HelloWorld!");
                sSPBasic->ReleaseSuite(kADMBasicSuite, kADMBasicSuiteVersion);
            }
        }
    }
    return error;
}
```

This code snippet is intended to show only the concepts of acquiring, using, and releasing a suite; this is not the preferred way to acquire and release suites. To see how suites normally are acquired and released, see the sample plug-ins provided in the SDK.

## Publishing suites

All plug-ins use suites, since they are the fundamental mechanism of the Illustrator API. Plug-ins also can publish suites of their own, for use by other plug-ins. This feature, where plug-ins may be both clients of suites and publishers of suites, is extremely powerful. Several plug-ins included with Illustrator (including all in the `Required` folder) publish suites used by many other plug-ins.

From the plug-in's perspective, it is unimportant whether a particular suite is implemented within Illustrator itself or as a plug-in. The Illustrator plug-in manager is responsible for managing suites among various plug-ins and the application.

To export a suite, a plug-in must call `SPSuitesSuite::AddSuite` on start-up. We also recommend that a plug-in declare a `PIExportsProperty` in its PiPL, to allow Illustrator to optimize the plug-in initialization process. Illustrator always tries to load plug-ins with a `PIExportsProperty` first, so other plug-ins that depend on exported suites can load and start up successfully on the first try.

For more information on how to publish suites, see `SPSuitesSuite::AddSuite` in *Adobe Illustrator API Reference* and ["Export property" on page 33](#).

## Binary compatibility

Illustrator makes its best effort to support binary compatibility across two previous product versions. For example, this means a plug-in built with the Illustrator CS1 SDK or Illustrator CS2 SDK should load and run under Illustrator CS3. Changes to the development, operating system, or hardware platform can compromise this objective. The compatibility of older plug-ins with Illustrator CS3 is described in the following table.

Product/SDK	Illustrator CS1 SDK plug-in	Illustrator CS2 SDK plug-in	Notes
Illustrator CS3 on Windows	Compatible	Compatible	Plug-ins built with the Illustrator CS1 SDK or Illustrator CS2 SDK are compatible with Illustrator CS3. In theory, binary compatibility on Windows can stretch back to Illustrator 7.0 for plug-ins that do not depend on the old text APIs that were obsoleted at Illustrator CS1. In practice, however, support for binary compatibility is limited to at most two previous product versions.
Illustrator CS3 on Macintosh	Incompatible	Compatible	<p>On a PowerPC Macintosh or Intel Macintosh under Rosetta, binary compatibility goes back to Illustrator CS2. Plug-ins built with the Illustrator CS2 SDK are compatible with Illustrator CS3; plug-ins built with the Illustrator CS1 SDK are not. The incompatibility is caused by the switch from PEF format to Mach-O format for plug-ins at Illustrator CS2.</p> <p>An Intel Macintosh Illustrator plug-in must be built as a Universal Binary, to be compatible with both Illustrator CS2 (a PowerPC Macintosh application) <i>and</i> Illustrator CS3 (a Universal Binary Macintosh application).</p>

## Illustrator's artwork as seen by a plug-in

Most plug-ins manipulate Illustrator artwork (including paths, text, and raster art) in some manner. Illustrator artwork objects are presented to plug-ins as a hierarchical tree, which plug-ins can freely modify.

Plug-ins can create, remove, and change the characteristics of artwork objects. For example, plug-ins can group objects, move or distort paths, adjust colors, and search and change text.

## Cross-platform support

The implementation of the Illustrator API is highly portable across platforms. Plug-ins can be written for Macintosh and Windows platforms. Working with Illustrator data types is the same on both platforms. Differences are related to the architectures of the hardware or operating system on which Illustrator runs, and these are abstracted, so the API call works on both environments with a minimum of platform support code. There are platform-specific API functions, but these exist largely for backward compatibility with earlier implementations; there are cross-platform versions that we suggest you use instead.

Because of this high level of compatibility, writing an Illustrator plug-in for Macintosh and Windows is fairly easy. The Illustrator-specific code can be run with little modification, often only a few lines of code within an `#ifdef`. For example, the Tutorial sample in the SDK uses identical source code; only the resources are platform specific.

The main differences are in presenting the user interface and resource data; depending on user-interface complexity, this can be a significant undertaking. User-interface items like menus are implemented using

the Illustrator API and are highly compatible across platforms. Controls within windows and tab panels present a larger problem, but the Illustrator API supports the ADM API for doing cross-platform user-interface work. ADM works for creating floating windows and modal dialogs. Platform-native dialog resources are used to define the user interface, but the code implementing it is cross platform. For more information on ADM, see *Adobe Dialog Manager CS3 Programmer's Guide*.

## Memory

Memory-management functionality is provided by `SPBasicSuite` and `SPBlocksSuite`. The Basic suite (`SPBasicSuite`) memory allocation functions are convenient since this suite is readily available when a plug-in is called (see `SPMessageData`).

## Resources

Illustrator plug-ins define their resources in the format native to the platform on which they are running. The resources can be accessed from a plug-in using `SPAccessSuite::GetAccessInfo`.

## Byte information and structures

Illustrator-specific data structures for the Macintosh and Windows implementations are the same, with the exception of platform dependencies like byte order. On Windows and Macintosh, byte alignment is to four-byte boundaries.

## System requirements

The platforms supported for Illustrator plug-in development are documented in *Adobe Illustrator CS3 Porting Guide*. Platform requirements for Illustrator are defined in the product release notes.

## Helper classes

The API provides helper classes that make it easier to write plug-in code. These classes encapsulate calling of suites; they remove the need to call suites directly in your code. For example, `ai::UnicodeString` provides support for Unicode strings and removes the need to call `AIUnicodeStringSuite`.

Source files for these helpers are provided by the API. To use a helper, add the `.cpp` file that implements it to your project and build the code as part of your plug-in. For example, `IAIUnicodeString.cpp` file implements the `ai::UnicodeString` class.

**NOTE:** Using helper classes is different than using suites. Suites are implemented by another binary component (the Illustrator application or its plug-ins) and called by your plug-in; an API header file defines the interface. Helper classes, on the other hand, are built as part of your plug-in; an API header file defines the interface, and an API source file (`.cpp` file) provides the implementation.

To use a helper class, a plug-in typically must provide pointers to the suites used by the class, in global variables with well-known names. For example, `ai::UnicodeString` requires that the calling plug-in acquire a pointer to `AIUnicodeStringSuite` and `SPBlocksSuite` in the global variables below:

```
extern "C" {
    AIUnicodeStringSuite* sAIUnicodeString = nil;
    SPBlocksSuite* sSPBlocks = nil;
}
```

If you add a helper class to your plug-in and you do not declare the global suite pointers it requires, you will get linker errors when you build your project. If you do not acquire the suites the class depends on, run-time errors will occur (see [“Acquiring and releasing suites” on page 16](#)). On Windows, if you add a helper class to your plug-in and your Visual Studio project is using pre-compiled headers, you will get a compilation error, because the source file for the helper class does not `#include` your pre-compiled header file. To fix this override the Create/Use Precompiled Header setting to be Not Using Precompiled Headers for the helper-class source file.

For documentation on the classes provided, see *Adobe Illustrator API Reference > Helper classes*.

## Plug-in adapters

Support for older plug-ins, as well as for Photoshop plug-ins, is provided through *plug-in adapters*, plug-ins that map API calls between Illustrator and the older or non-native APIs of the other plug-ins. Adapters are not discussed in detail in this document.

## About Plug-ins menu

To display company contact information or copyright statements about your plug-in, follow these steps:

- Create a new menu group under Illustrator’s About group (see `kAboutMenuGroup`), to contain all the About plug-in menu items.
- For each plug-in, create a menu item under this new group.
- Handle the menu message related to use of your About plug-in menu by popping an About box containing plug-in specific information.

An illustration of the resulting menus on Mac OS is given below:

```
Illustrator > About MyCompanyName Plug-ins > MyPluginName1...
                                                MyPluginName2...
```

An illustration of the resulting menus on Windows is given below::

```
Help > About MyCompanyName Plug-ins > MyPluginName1...
                                         MyPluginName2...
```

A helper class that supports this functionality is provided as sample code; see `SDKAboutPluginsHelper`. The Tutorial sample shows how to use this class.

**NOTE:** Before Illustrator CS3, the list of plug-ins that were loaded was displayed in the About Plug-ins dialog, and the user could choose to request further information about a specific plug-in. As of Illustrator CS3, the About Plug-ins dialog has been removed.

## Next steps

In this chapter, we introduced plug-ins for Adobe Illustrator, and defined several key concepts, including:

- Plug-in manager
- Plug-in types
- Messages and message actions

- Notifiers
- Loading and unloading
- Acquiring, using, and releasing suites

[Chapter 2, "Tutorial"](#) explains the process of writing a plug-in in more detail. *Adobe Illustrator API Reference* describes each function in detail and provides information on the artwork objects and data structures you need to complete the job. Finally, the sample projects in the SDK provides concrete examples and are a good starting point for your project.

## 2 Tutorial

This chapter describes the fundamentals of Illustrator plug-in programming, using as an example a plug-in called Tutorial that is provided on the SDK. Look for the sample project in the sample code folder on the SDK, and browse the source code while reading this chapter.

The core characteristics of an Illustrator plug-in were introduced in [Chapter 1, “Overview](#). If you have not read that chapter already, do so before proceeding through this chapter.

### PiPL resource and plug-in entry point

Illustrator plug-ins must have a valid PiPL resource and an entry point Illustrator can call.

The Tutorial plug-in's PiPL is defined in the following source files:

- Windows: See the PiPL resource declaration in `Tutorial.rc`.
- Macintosh: See the PiPL resource declaration in `Tutorial.r`.

For more information on PiPLs, see [“Plug-in Property Lists” on page 31](#).

The Tutorial plug-in's entry point is the `PluginMain` function in `Tutorial.cpp`.

Only plug-ins with the Illustrator version information in the PiPL are recognized and included in Illustrator's initial start-up process. A plug-in is loaded into and unloaded from memory as needed by Illustrator. Your plug-in should be written assuming it is not always in memory. This is why your plug-in should save and restore state information during unload and reload. Your plug-in can expect certain services from the application. Because your plug-in may be unloaded, Illustrator provides a means of storing important data when the plug-in is unloaded. Each time the plug-in is called, it is given enough information to accomplish the action to be performed.

### Handling callers and selectors

Your plug-in's organization is based largely on the messages received by its `PluginMain` function. The main routine of your plug-in basically becomes a switch implemented as a series of string comparisons that call functions appropriate for the message action caller and selector. See [“Handling callers and selectors” on page 14](#).

### Using suites and callback functions

The Illustrator API provides the core functionality used by a plug-in. Illustrator's API comprises callback functions organized into suites. Before your plug-in can use a function that is part of a suite, the suite containing it must first be *acquired*. A suite is a structure filled with function pointers; when your plug-in acquires a suite, a pointer to this structure is returned.

When the suite is no longer needed, the acquired suite is *released*. It is important to do this, so the Illustrator plug-in manager can run optimally. For instance, Illustrator keeps track of how many times a suite was acquired. If a suite added by plug-in is no longer in use (its reference count is 0), the plug-in may be unloaded to free memory.

## Acquiring and releasing suites and calling suite functions

When your plug-in is first called, it knows about only one suite. The message data structure passed to all plug-ins has a member variable named *basic*, which points to the Basic suite (see `SPBasicSuite`). The Basic suite is used to access other suites and contains two important functions for doing so:

```
SPAPI SPError (*AcquireSuite)( const char *name, long version, const void **suite );
SPAPI SPError (*ReleaseSuite)( const char *name, long version );
```

A plug-in uses the first function, `SPBasicSuite::AcquireSuite`, to gain access to a suite of functions. All acquired suites must be released with the `SPBasicSuite::ReleaseSuite` function when the suite is no longer required.

To acquire a suite, you first need to declare a suite pointer. Then you call `SPBasicSuite::AcquireSuite`, using the name and version number of the suite you want, found in its public header file. For instance, suppose you need to use Illustrator's Menu suite, `AIMenuSuite`:

```
AIMenuSuite* sAIMenu = nil;
SPBasicSuite* sSPBasic = ( ( SPMessageData* )message)->basic;
SPError error = sSPBasic->AcquireSuite(kAIMenuSuite, kAIMenuSuiteVersion, &sAIMenu);
```

A pointer to the acquired suite is returned in `sAIMenu`, and functions in the suite can be called through this pointer:

```
long count = 0;
sAIMenu->CountMenuItems(&count);
```

After the function is used, the suite is released:

```
sSPBasic->ReleaseSuite( kAIMenuSuite, kAIMenuSuiteVersion );
```

Since they are used throughout the plug-in code, it is convenient to make suite pointer variables global. The convention used for these global variables is a small "s," followed by the owner of the suite, followed by the suite name; e.g., `sSPBasic` as shown above, `sAIMenu` for the Menu suite, etc.

## Illustrator API suites

Every suite has the suite name and version in the suite header file, along with other definitions, like error strings, that are particular to their function. If the suite defines plug-in messages, they also are in the header file, with the suite functions. The function pointers are fully prototyped.

Full documentation on available suites and the functions they contain is in *Illustrator API Reference*.

## Callers and selectors

This section describes and illustrates what the various caller/selector pairs mean and what your plug-in is expected to do in response to receiving them.

### Caller: `kSPInterfaceCaller`, selector: `kSPInterfaceStartupSelector`

When Illustrator starts up, each plug-in receives a `kSPInterfaceStartupSelector` message. Only one such message is received per Illustrator session. This is where the plug-in needs to allocate memory and add plug-in types to Illustrator. Sample code that handles start-up in the Tutorial plug-in looks something like this:

```

typedef struct {
    AIFilterHandle    filterVariation1;
    AIToolHandle      toolVariation1;
    AIMenuItemHandle aboutPluginMenu;
} Globals;

Globals* g = nil;

static AIErr StartupPlugin ( SPInterfaceMessage* message )
{
    AIErr error = kNoErr;
    error = AcquireSuites( message->d.basic );
    if (!error) {
        // Allocate our globals - Illustrator will keep track of these.
        error = message->d.basic->AllocateBlock( sizeof(Globals), (void **) &g );
        if ( !error ) {
            message->d.globals = g;
        }
    }
    if (!error) {
        error = AddFilter(message);
    }
    if (!error) {
        error = AddTool(message);
    }
    if (!error) {
        error = AddAction(message);
    }
    if (!error) {
        error = AddMenu(message);
    }
    ReleaseSuites( message->d.basic );
    return error;
}

```

When allocating memory during this phase, you should use memory management functions provided by the Illustrator API (see `SPBasicSuite` or `SPBlocksSuite` for example) and put a reference to the memory in the `globals` field of the `SPMessageData` structure. Illustrator keeps this value for you and passes it back to you on subsequent calls, so it is a convenient place to store information you may need next time the plug-in is loaded. Illustrator does not care what you put in the `globals` field. Usually, it is a pointer to a block of memory. If you do not have any global data, you can leave the `globals` field empty.

During the start-up message, you need to inform Illustrator of the filters, tools and so on that your plug-in adds. For example, the `StartupPlugin` function above adds a filter. The filter is added by calling the `AIFilterSuite::AddFilter` function provided by the Illustrator API. See the `AddFilter` function in `Tutorial.cpp` for the code that makes this call:

```

error = sAIFilter->AddFilter( message->d.self, "Tutorial", &filterData,
                             kToolWantsToTrackCursorOption, &g->filterVariation1);

```

When adding most plug-in types, the same or similar arguments are used. The first argument is a reference to the plug-in itself. You can get this from the `SPMessageData` structure. The second argument is an identifier for the plug-in type. This C-style string identifies the current added plug-in to a second plug-in, one perhaps searching for the plug-in's functionality. To be helpful, it should be as descriptive as possible.

Following the identifier are data and options specific to the plug-in type. The data is any information specific to the plug-in type on a platform. For instance, the filter above has a filter *category* and filter *title*. All filters with the same category are placed together in one submenu.



The options specify plug-in behavior to be provided by Illustrator. Filters do not have any special behavior but could use some default behavior options, like the `kPluginWantsResultsAutoSelectedOption` constant, which is used to control how artwork is selected when the plug-in returns control to Illustrator.

The final argument is returned to the plug-in by Illustrator. It is a reference to the added plug-in type. For simple plug-ins, where only one instance of a given plug-in type is added, this value can be ignored. If a plug-in adds more than one instance of a type, this reference should be saved in the `globals` block whose reference is returned to Illustrator. When a plug-in is called, Illustrator passes the active instance of the plug-in inside the message data. The saved references and current plug-in are compared later, to determine which plug-in instance was selected by the user:

```
if ( (AIFilterMessage)message->filter == g->filterVariation1 ) {
    // Do something for this variation
}
else {
    // Do the other variation
}
```

Of course, if you have no special requirements, you do not need to do any checking. That is it for initialization. The `kSPInterfaceStartupSelector` selector is called only once.

## Caller: `kSPInterfaceCaller`, selector: `kSPInterfaceShutdownSelector`

When Illustrator is in the process of quitting, each plug-in receives a `kSPInterfaceShutdownSelector` message. Only one such message is received per Illustrator session. Actions that should happen when the user is completely finished using the plug-in, like saving preference information, are done at this time. Also, any needed follow-up action for something done during the `kSelectorAIStartupPlugin` message should be done now. Some actions, like adding a plug-in type, do not need any clean up. A common example is freeing allocated memory that the system does not automatically free when the application is quit. A shut-down routine corresponding to the `StartupPlugin()` function above would look like this:

```
static AIErr ShutdownPlugin( SPInterfaceMessage* message )
{
    AIErr error = kNoErr;
    if ( g != nil ) {
        message->d.basic->FreeBlock(g);
        g = nil;
        message->d.globals = nil;
    }
    return error;
}
```

## Caller: `kSPAAccessCaller`, selector: `kSPAAccessReloadSelector`

Plug-in code is loaded and unloaded dynamically, depending on whether Illustrator is using it. If the code is unused by the main application or another plug-in for a pre-defined period of time, it is unloaded. This is true of all plug-in types, including plug-in suites. Illustrator notifies the plug-in of the loading and unloading events.

After the start-up selector is received, each time your plug-in is brought into memory, it receives the `kSPAAccessCaller/kSPAAccessReloadSelector` message pair. Reload is your plug-in's opportunity to restore state information it needs to run, like global variables. Plug-in suites use the reload message to set up their function tables. A reload routine looks something like this:

```
Globals* g;

static AIErr ReloadPlugin( SPMessages *message )
{
    AIErr error = kNoErr;
    g = ( Globals* )message->d.globals;
    return error;
}
```

## Caller: kSPAccessCaller, selector: kSPAccessUnloadSelector

The opposite of the reload selector is `kSelectorAIUnloadPlugin`. This is an opportunity for the plug-in to save any state information before being removed from memory. Here is a sample routine for the unload selector:

```
static AIErr UnloadPlugin( SPMessages* message )
{
    AIErr error = kNoErr;
    message->d.globals = g;
    return error;
}
```

## Plug-in type-specific messages

The selectors discussed so far are received by all plug-ins. Other caller/selector pairs a plug-in receives depend on the *plug-in types* added at start-up. This section describes the caller/selector pairs associated with the plug-in types used in the Tutorial. For a description of the major plug-in types Illustrator supports, see [Chapter 1, “Overview.”](#)

## Filter plug-ins

Illustrator allows plug-ins to add new filters to the Filters menu. To add a filter, your plug-in must do the following:

- Call `AIFilterSuite::AddFilter` on start-up, to add the filter to Illustrator.
- Handle messages relating to filter events.

There are two caller/selector pair messages associated with Filter plug-in types:

- Get-filter parameters (`kCallerAIFilter / kSelectorAIGetFilterParameters`).
- Go filter (`kCallerAIFilter / kSelectorAIGoFilter`).

The get-filter-parameters selector pair is an opportunity for your plug-in to present a dialog to the user, requesting information about how the plug-in should work. It is followed by the go-selector pair, which is when the plug-in actually does its work. The parameters are acquired in a separate call to support the Last Filter feature in the Filter menu, which applies the last-used filter without asking the user for a new set of parameters. In this case, you receive a `kCallerAIFilter/kSelectorAIGoFilter` pair without a preceding `kCallerAIFilter/kSelectorAIGetFilterParameters` pair.

The Tutorial plug-in's `PluginMain` function handles these selector pairs as follows:

```

    } else if ( strcmp( caller, kCallerAIFilter ) == 0 ) {
        if ( strcmp( selector, kSelectorAIGetFilterParameters ) == 0 )
            error = GetFilterParameters( ( AIFilterMessage* )message );
        else if ( strcmp( selector, kSelectorAIGoFilter ) == 0 )
            error = GoFilter( ( AIFilterMessage* )message );
    }
}

```

When the user chooses your filter in the Filter menu, your plug-in is first called with `kSelectorAIGetFilterParameters`. You should get whatever parameters you need from the user and place them in a handle, storing the handle in the `parameters` field of `AIFilterMessage`. This is similar to how the `globals` field is used, only `parameters` must be a handle if it is not nil. The `parameters` handle is given back to you on subsequent calls; therefore, it is necessary to create this handle only once. Each time you receive the `get-filter-parameters` selector, you can use the previous `parameters` to set the default dialog values. If your plug-in places more than one filter in the Filter menu, a separate `parameters` handle is kept for each one. Illustrator does not care about the specific contents of the `parameters` handle, but the handle must be self contained; that is, you cannot include pointers or handles to more data inside the `parameters` handle. This is because Illustrator may make copies of the `parameters` and does not know how to copy blocks of memory referenced by the structure. If you do not have any parameters, leave this field zero.

The parameters used by the Tutorial plug-in's filter are shown below:

```

typedef struct {
    ASInt32 corners;    // number of corners on the polygon
    ASInt32 size;       // the maximum size, in points
} MyFilterParameters;

```

These parameters are obtained from the user by the `GetFilterParameters` function in `Tutorial.cpp`. The return result from `GetFilterParameters` should be `kNoErr` if the user hit OK (or if you do not have any parameters) or `kCanceledErr` if the user cancels the filter.

Immediately after the plug-in returns from `GetFilterParameters`, it is sent the `go-filter` selector pair (`kCallerAIFilter/kSelectorAIGoFilter`). When the plug-in receives this, it performs whatever the filter does, using the `globals` and `parameters` the plug-in set up previously. The Tutorial plug-in's `GoFilter` function in `Tutorial.cpp` creates a random polygon each time it is called, by creating a new path art object and adding path segments to it.

## Tool plug-ins

Illustrator allows plug-ins to add new tools to the Tools panel. To add a tool, your plug-in must do the following:

- Provide an icon to represent the tool in the Tools panel.
- Call `AIToolSuite::AddTool` on start-up, to add the tool to Illustrator.
- Handle tool-related messages relating to mouse events. For example, when the plug-in's tool is selected in the Tools panel and the mouse button goes down and is released, the plug-in is notified by a selector message (`kCallerAITool/kSelectorAIToolMouseDown`). A similar approach is used to communicate mouse drag events, and so on.

The Tutorial plug-in adds a line-drawing tool by calling `AIToolSuite::AddTool` on start-up. For sample code, see the `AddTool` function in `Tutorial.cpp`.

The added tool appears in the Tools panel. When it is selected and used, Illustrator calls the plug-in with the tool selectors listed above. To create a minimal straight-line tool, the plug-in needs to handle only the mouse-down selector. The Tutorial plug-in's `PluginMain` function handles this as follows:

```
else if ( strcmp( caller, kCallerAITool ) == 0 ) {
    if ( strcmp( selector, kSelectorAIToolMouseDown ) == 0 )
        error = ToolMouseDown( ( AIToolMessage* )message );
}
```

When the mouse is clicked, the Tutorial plug-in receives this selector and can process it. See the `ToolMouseDown` function in `Tutorial.cpp`. This function draws path segments to each mouse-down location. The location of the mouse is passed in the tool-message structure; see `AIToolMessage` in *Adobe Illustrator API Reference*. The tool message contains the basic plug-in data and a reference to the tool being used. The `cursor` field contains the point on the art board where the last tool event occurred, and the Tutorial plug-in's mouse-down handler uses this to create a path. The `ToolMouseDown` function begins by acquiring the suites it needs to make a line. The function checks whether a selected path exists. If so, it adds path segments to it; otherwise, it creates the first point in a new path. The function ends by releasing the suites it acquired.

While the `ToolMouseDown` function adds a basic tool, other selectors can be processed to give the line tool more functionality, like setting the cursor or tracking a mouse drag. This is left as an exercise for the reader.

## Action plug-ins

For your plug-in to be recordable by Illustrator's Actions panel, you must add special code to your plug-in. Follow these steps:

1. **Register action events** — During start-up, your plug-in must register one or more action events. An action event is a single operation your plug-in executes. An Action event is shown in Illustrator's Actions panel, if the user chooses to record it.
2. **Record action events** — During your plug-in's execution, you must check whether the user is in record mode. If so, you must record your action event and pass all relevant parameters to the Action Manager.
3. **Respond to the DoAction selector** — Once you register one or more Action Events, your plug-in must be ready to execute those Action Events when requested. Your plug-in must respond to the `kActionCaller` caller and `kDoActionSelector` selector. These are defined in the Action Manager suite header file.

## Registering action events

First, determine how many action events your plug-in will register, by separating the functionality of your plug-in into discrete operations. Basically, try to think of which operations users will want to record in the Actions panel.

During start-up, your plug-in should first make sure the Action Manager suite (see `AIActionManagerSuite`) is available, by trying to acquire a pointer to the suite. This check is necessary because the Action Plug-in may be disabled by removing it from the `Plug-ins` folder.

To register an action event, your plug-in must do the following:

- Call `AIActionManagerSuite::AINewActionParamType` to create a type parameter block (TPB) that describes the parameters your action event requires.

- Populate the TPB with key-value pairs that give the name and data type of each parameter by calling `AIActionManagerSuite::AIActionSetTypeKey`.
- Call `AIActionManagerSuite::RegisterActionEvent` to register the action event.
- Call `AIActionManagerSuite::AIDeleteActionParamType` to dispose of the TPB.

For sample code, see the `AddAction` function in `Tutorial.cpp`.

## Recording action events

During your plug-in's execution, if the user is in record mode, you are responsible for recording any action events you registered. Illustrator is in record mode when the user is recording actions via the Actions panel.

To record an action, your plug-in must do the following:

- Call `AIActionManagerSuite::InRecordMode` to check if actions are being recorded.
- Call `AIActionManagerSuite::AINewActionParamValue` to create a Value Parameter Block (VPB) in which parameter values are recorded. A VPB is different from a TPB, in that it can contain actual values. A TPB can contain only data type descriptions.
- Populate the VPB with key value pairs that give the name and value of each parameter, by calling the appropriate `AIActionManagerSuite` function for the type of data stored in the parameter. See `AIActionManagerSuite::AIActionSetBoolean`, `AIActionManagerSuite::AIActionSetInteger`, etc.
- Call `AIActionManagerSuite::RecordActionEvent` to record the action event.
- Call `AIActionManagerSuite::AIDeleteActionParamValue` to dispose of your reference to the VPB.

For sample code, see the `GoFilter` and `RecordFilterAction` functions in `Tutorial.cpp`.

## Responding to the DoAction selector

To execute an action, your plug-in must handle the do-action selector pair (`kActionCaller/kDoActionSelector`) for each action event the plug-in registers. This is Illustrator's way of requesting that a particular action event be executed.

Your plug-in must add code for detecting such a notification. The Tutorial plug-in's `PluginMain` function handles the action selector, as follows:

```
else if ( strcmp( caller, kActionCaller ) == 0 ) {
    if ( strcmp( selector, kDoActionSelector ) == 0 )
        error = DoAction( ( DoActionMessage* )message );
}
```

The `DoAction` function in `Tutorial.cpp` executes the requested action event. The message struct sent along with the `kDoActionSelector` is `DoActionMessage`. Documentation for this structure is in *Adobe Illustrator API Reference*. It contains the recorded parameter values the action must replay. The `DoAction` function extracts the parameter values into its own data structure. The `DoActionMessage` struct also indicates whether a dialog should be popped to allow the user to tune the parameter values before the action is executed. The `DoAction` function responds accordingly, then calls the Tutorial plug-in's filter function, `GoFilter`, to re-play the action.

## Playing action events

You also can play back action events from a plug-in. These action events could originate in the Illustrator application or other plug-ins. The `SnpDocumentActionHelper` code snippet in the `SnippetRunner` plug-in demonstrates how to play action events that originate in the Illustrator application.

## Menu Plug-ins

Illustrator allows plug-ins to add new menus to its menu structure. To add a menu, your plug-in must do the following:

- Call `AIMenuSuite::AddMenuItem` on start-up, to add the menu to Illustrator.
- Optionally, call `AIMenuSuite::AddMenuGroupAsSubMenu` to create a group in which further menu items can be nested.
- Handle messages related to menu events. For example, when a plug-in's menu item is clicked, the plug-in is notified by a selector message (`kCallerAIMenu/kSelectorAIGoMenuItem`).

The Tutorial plug-in adds an About plug-in menu on start-up; see [“About Plug-ins menu” on page 20](#). For sample code, see the `AddMenu` function in `Tutorial.cpp`. A helper class, `SDKAboutBoxHelper`, is used to create the menu that appears under Illustrator's About menu group. When it is used, Illustrator calls the plug-in with the tool selectors listed above. The Tutorial plug-in's `PluginMain` function handles this as follows:

```
else if ( strcmp( caller, kCallerAIMenu ) == 0 ) {
    if ( strcmp( selector, kSelectorAIGoMenuItem ) == 0 )
        error = GoMenu( ( AIMenuMessage* )message );
}
```

When the menu is clicked, the Tutorial plug-in receives this selector and can process it. See the `GoMenu` function in `Tutorial.cpp`. This function pops an About box that displays contact details and a copyright statement.

## Next steps

To learn more about Illustrator plug-in programming, explore the samples provided in the SDK and the documentation in *Adobe Illustrator API Reference*. Instructions on running and debugging plug-ins are in *Adobe Illustrator CS3 Porting Guide*.

## 3 Plug-in Property Lists

A PiPL resource contains a list of properties that store information about a plug-in. Illustrator considers only those files with valid PiPL resources to be potential plug-ins.

### PiPL samples

Sample PiPL resource files are provided on the SDK in source code and binary form.

All sample plug-ins in the SDK define their PiPLs in a resource source-code form. For example, the Tutorial plug-in's PiPL is defined in the following source files:

- Windows — See the PiPL resource declaration in the `Tutorial.rc` file.
- Mac OS — See the PiPL resource declaration in the `Tutorial.r` file.

As the samples show, it is more flexible to work with PiPL resources in source-code form. Alternately, binary PiPL files provided in the SDK can be added to your projects, to create a plug-in that will be recognized and loaded by Illustrator; see the following table.

File	Platform	Notes
PiPL.bin	Windows	This binary file contains a basic PiPL resource. The file is in the <code>&lt;SDK&gt;/samplecode/common/win/</code> folder. To use this file, add the line below to your plug-in's <code>.rc</code> file:  <code>16000 PIPL DISCARDABLE "...\..\Common\Win\PiPL.bin"</code>
PiPL.rsrc	Mac OS	This binary file contains a basic PiPL resource that can be added directly to your plug-in project. The file is in the <code>&lt;SDK&gt;/samplecode/common/mac/</code> folder.

### PiPL structure

A plug-in property list has a version number and count, followed by a sequence of arbitrary-length byte containers called *properties*. The core types that define a PiPL are documented in the *Illustrator API Reference* and listed in the following table:

Type	Note
PIProperty	A plug-in property. Each property has a vendor ID, key, ID, length, and property data (whose size is indicated by the property length). The vendor ID identifies the vendor that defined the property type. All PiPL properties defined by the Illustrator API use a vendor ID of ADBE. Each property must be padded such that the next property begins on a four-byte boundary.
PIPropertyList	A plug-in property list has a version number and count, followed by a sequence of arbitrary-length byte containers called properties.

## Required PiPL properties

Your plug-in's PiPL resource must include the required properties listed in the following table.

Property key		Value	Description
<code>PIKindProperty</code>	<code>kind</code>	<code>SPEA</code>	Indicates the type of the plug-in file; it is akin to a file type. Illustrator loads plug-ins whose kind property have the value <code>SPEA</code> .
<code>PISPVersionProperty</code>	<code>ivrs</code>	<code>2</code>	Describes to Illustrator the calling conventions expected by the plug-in and currently has a value of 2.

Your plug-in also must have at least one *code-descriptor* property that tells Illustrator the entry point of your code. Code descriptors are available for Intel-based Macintosh, PowerPC-based Macintosh, and Windows-based plug-ins:

Property key		Description
<code>PICodeMacIntel32Property</code>	<code>mi32</code>	<code>PIMacIntelCodeDesc</code> code descriptor containing entry point of Intel code for plug-ins on Macintosh platforms.
<code>PIPowerPCMachOCodeProperty</code>	<code>mach</code>	<code>PIMachCodeDesc</code> code descriptor containing entry point of PowerPC code in Mach-O format for plug-ins on Macintosh platforms.
<code>PIWin32X86CodeProperty</code>	<code>wx86</code>	<code>PIWin32X86CodeDesc</code> code descriptor containing entry point of plug-ins on Windows platforms.

Your plug-in binary can contain multiple code-descriptors if it will run on different types of machines. For example, a universal binary Macintosh plug-in contains Intel and PowerPC code descriptors.

## Optional PiPL properties

Your plug-in's PiPL resource can include the optional properties described in the following table.



Property key	Notes
PIExportsProperty <i>expt</i>	<p>Plug-ins can export one or more suites containing functions for use by other plug-ins. Suites are discussed in <a href="#">“Suites” on page 15</a>.</p> <p>To ensure the interdependencies of plug-ins are handled correctly, plug-ins declare in advance what they export. The loading order of plug-ins becomes important when one plug-in depends on a suite provided by another plug-in.</p> <p>This export information is declared in PiPL export properties, <i>expt</i>, which contain the names and API version numbers of the suites a plug-in provides. The plug-in manager uses this information to load and execute plug-ins, ensuring that suites and other resources are available when needed.</p>
PIPluginNameProperty <i>pinm</i>	The plug-in name displayed in Illustrator’s Help > System Info dialog.

## Export property

Plug-ins can extend the functionality of the API by exporting new suites. To optimize Illustrator’s plug-in initialization process, a plug-in should add an export property (PIExportsProperty) to its PiPL. See the following table.

Property key	Notes
PIExportsProperty <i>expt</i>	<p>Plug-ins can export one or more suites containing functions for use by other plug-ins. This export information is declared in PiPL export properties, <i>expt</i>, which contain the names and API version numbers of the suites a plug-in provides. The plug-in manager uses this information to load and execute plug-ins, ensuring that suites and other resources are available when needed. When a suite is requested (acquired), the export properties of all plug-ins are searched for the suite, and the providing plug-in is loaded into memory. The plug-in is started if necessary. Once the plug-in providing the suite is loaded and the requested suite is available, control is returned to the requesting plug-in. One suite request could trigger a series of plug-ins to be loaded into memory in a cascading fashion.</p>

The property data that describes the suites that are to be exported is represented in the form of C structs, below. This representation must be transcribed into a resource source code form (Rez resources (.r) on Macintosh or Windows resources (.rc) under Visual Studio):

```
/** List of suites exported by the plug-in. */
typedef struct MyExportsDesc
{
    /** The number of suites exported by the plug-in. */
    long fCount;
    /** A variable-length list describing each suite exported by the plug-in. */
    MyExportDesc fExports[1];
} MyExportsDesc;

/** Description of a suite exported by the plug-in. */
typedef struct MyExportDesc
{
    /** The total length in bytes of this MyExportDesc record.*/
    long fLength;
    /** A C-style string with the name of the suite to be exported.
     * Padded to 4 bytes. */
    char fName[1];
    /** The version of the suite to be exported. */
    long fVersion;
} MyExportDesc;
```

## For more information

*Adobe Illustrator API Reference* describes the property names and associated data structures, such as `PIPropertyList`. The sample plug-ins on the SDK show how to define a PiPL using native platform resources.

# Index

## A

- about message, 13
- Acquiring Suites, 23
- acquiring suites, 16
- Action, 8
- Action Plug-ins, 28
- Adobe Dialog Manager, 6

## C

- Callers, 12, 14
- code-descriptor, 32

## D

- DoAction Selector, 29

## E

- Effects, 8
- expt, 33

## F

- File Format, 8
- Filter, 8
- Filter Plug-ins, 26

## G

- globals (SPMessageData), 15

## I

- ivrs, 32

## K

- kind, 32
- kSPAccessCaller, 12
- kSPAccessReloadSelector, 12
- kSPAccessUnloadSelector, 12
- kSPInterfaceCaller, 13
- kSPInterfaceShutdownSelector, 13
- kSPInterfaceStartupSelector, 13

## L

- loading order, 11

## M

- mach, 32
- Menu Command, 8
- message action, 12
- Message Data, 14
- Messages, 11
- mi32, 32

## N

- Notifier, 8
- Notifiers, 14
- notifiers, 14

## P

- PICA (Plug-in Component Architecture), 16
- pinm, 33
- PiPL, 11
- PiPL resources, 11
- Plugin Group, 8
- publishing suites, 17

## R

- Releasing Suites, 23
- releasing suites, 16
- reload message, 13

## S

- Selectors, 12, 14
- shutdown message, 13
- SPMessageData structure, 14
- startup message, 13
- Suite, 8
- suites
  - acquiring and releasing, 16
  - calling suite functions, 20
  - overview, 15
  - publishing, 17

## T

Timer, 8

Tool, 8

Tool Plug-ins, 27

Transform again, 8

## U

unload message, 13

## W

wx86, 32