Adobe® Photoshop® 6.0 Automation Tutorial

Version 6.0 Release 1
August 2000

Photoshop Automation Plug-in Tutorial

This document was written by Tina Wu. Some material based on documents written by Chris Bailey, Bobby Smith, Andrew Coven, Michael Snyder, John Long, and Sean Parent.

Version History

Date	Author	Status
1 June 2000	Tina Wu	First Draft.

1 Introduction

Audience

This document is intended for C/C++ programmers who want to automate Photoshop. You are expected to have a working knowledge of Adobe Photoshop and to understand how internal and plug-in features work from a user's point of view. This document assumes you understand Photoshop terminology such as paths, layers and masks. For more information, consult the Adobe Photoshop User Guide.

How to Use The Tutorial

Chapter 2 Automation Overview explains the concept of automation as a series of Photoshop "events" and defines the major components of an event
Chapter 3 Listener Plug-in explains building a function that incorporates the event components.
Chapter 4 Project Workspace provides step-by-step instructions to set up a project workspace for a Photoshop automation plug-in.
Chapter 5 Get Info Functions describes three example functions of getting information from Photoshop.

What The Tutorial Doesn't Have

The tutorial does not discuss creating a user interface nor creating scripting parameters. The MakeNew sample plug-in demonstrates these topics in the MakeNewUI.cpp and MakeNewScripting.cpp files.

2 Automation Overview

Introduction

This chapter explains the relationship of the automation plug-in to the Photoshop Actions palette, explain the concept of automation as a series of Photoshop "events", and define the major components of an event: the event name, the descriptors, and the reference. After reading this chapter, you will understand the relationship between the event components. The Listener chapter explains building a function that incorporates the event components.

Automation Plug-ins

Automation plug-ins are used to execute Photoshop menu commands in a programmatic fashion, much like that of the Photoshop Actions palette. The Actions palette provides a visual mechanism for users to record, store and replay sequences of Photoshop "events". An event is one or more Photoshop commands or instructions. Starting with Photoshop 4.0, Photoshop permitted users to manually automate Photoshop events in this fashion. However, with Photoshop 4.0 user interaction is required at each step of recording and playing. Starting in Photoshop 5.0, a new type of plug-in was created: the automation plug-in, which plays Photoshop events like the Actions palette, but at the programming level. Additionally, unlike previous plug-ins that were restricted to a simple interaction between Photoshop and the plug-in, an automation plug-in can interact with other plug-ins and any Photoshop event.

Events

An event is one or more Photoshop commands or instructions. Some examples of event names include: eventCrop, eventOpen, and eventSharpen. Usually, the name is descriptive of the event it represents. The event names are listed in PITerminology.h file of the SDK. Do not worry if you are not sure which event name(s) represents a specific Photoshop command. The Listener plug-in (included in the SDK) provides the event name(s) that represent the desired events. This is discussed in the Listener chapter.

Descriptors

A descriptor "describes" an event or an event parameter. Most Photoshop events require a descriptor. For example, before playing "eventGaussianBlur", you must specify the radius value in the descriptor of "eventGaussianBlur". In rare cases, a Photoshop event does not require a descriptor. Such is

the case with "eventPaste", an event which pastes the contents of the clipboard to the document.
A descriptor can also describe a parameter. For example, to make a rectangular-shaped selection, you need a descriptor for the rectangle, that defines the four corners of the rectangle. You must create a descriptor that specifies the four values of the rectangular selection. Making and specifying values in a descriptor are explained in the Listener chapter.

References

The reference specifies the object on which the event is played. Examples of objects include documents, layers, and the application, which are referred to as "classDocument", "classLayer", and "classApplication", respectively. Photoshop objects are listed in PITerminology.h file of the SDK. For example, you may want play an event on "classDocument". You need to specify "classDocument" as the reference. Sometimes, it is unnecessary to set the reference. By default, the reference is the current object. If the current object is not the object type that the event expects, Photoshop searches all of the objects starting with the current object until a container is found that can hold the object being manipulated. Setting the reference to a specific object is explained in the Listener chapter.

Conclusion

Up to this point, you should have a general idea of Photoshop events, and their major components: event names, descriptors and references. The Listener chapter discusses the code necessary to play Photoshop events with the aid of the Listener plug-in.

3 Listener Plug-in

Introduction

Before reading this chapter, you should have a general idea of Photoshop events and their components, including descriptors and references. If you do not have a general knowledge of Photoshop events, please read the previous chapter.
This chapter discusses the code to play Photoshop events. With the aid of the Listener plug-in, you can easily construct a function that calls a Photoshop event. The Listener plug-in is fundamental in creating C code to play Photoshop events. The first part of the chapter describes the Listener plug-in functionality, loading the Listener plug-in into Photoshop, and verifying that the Listener plug-in is working properly. The latter portion of the chapter introduces several samples of Listener C code output, explain each sample code line by line, and show a robust version of each of the functions.

What Is Listener?

The Listener is a sample automation plug-in designed to be a utility for automation plug-in development. Throughout the working session of Photoshop, the Listener plug-in silently "records" all the user actions in Photoshop. After each user-level event the Listener plug-in writes the C code that represents the event to a textfile, called "Listener.log". You can use the Listener output and copy and paste it directly into your plug-in source code. Not all events in Photoshop can be automated or "actionable". Some events that are actionable include selecting menuitems, selecting documents, and moving layers. For example, you can automate the procedure of selecting the Background layer and applying Gaussian Blur to that layer. Whereas, paintbrush strokes are not actionable.

Setting Up Listener Plug-In

You must create an alias/shortcut of the debug version of the Listener sample plug-in to the Photoshop "Plug-Ins" folder. For Mac, the Listener debug version is located at:
Adobe Photoshop 6.0 SDK:SampleCode:Output:Mac:Debug:Listener.8li
You must select the "Listener.8li" file. Create an alias to the "Plug-Ins" folder by, choosing from the menu "File" > "Make alias". Drag the alias to the Plug-ins folder of Photoshop. For Windows, find the the Listener debug version is located at:
Adobe Photoshop 6.0 SDK\SampleCode\Output\Win\Debug\Listener.8li
Right click on the Listener.8li file. Choose from the menu "Edit" > "Copy". Go to the Photoshop "Plug-ins" folder. Right click in this folder and select "Paste Shortcut".
Note: Do not use the Listener.8li located in Adobe Photoshop 6.0 SDK\SampleCode\Output\Win\Release or Adobe Photoshop 6.0 SDK:Sample-

Code:Output:Mac:Release. This version does not output C code to Listener.log

Is Listener Working?

To verify that the Listener plug-in is running properly, start by launching Photoshop. Photoshop loads all the plug-ins that are in the Photoshop "Plug-Ins" folder. When the Listener plug-in is loaded for the first time, it creates a Listener.log textfile. For Windows, the Listener.log file is located in C:\Listener.log. For Mac, Listener.log is located on the desktop. Second, check under the menu for "File" > "Automate" > "Listener" to verify the Listener plug-in is loaded.

The Automation Plug-In Development Process

You do not need to write any major source code to build an automation plug-in. The Listener plug-in provides the necessary source code for automating Photoshop events. First, you must manually execute user-level events for the Listener plug-in to record that you want your plug-in to execute. The Listener plug-in writes to the Listener.log file the corresponding C code, which is a series of function definitions. Essentially, the function definitions in Listener.log are complete. You must simply copy and paste the function definitions to your source file and make function calls to them from the PluginMain function. The MakeSelectFill sample plug-in includes function definitions that were created by the Listener plug-in and demonstrates how to call the functions from the PluginMain function.

Listener Plug-In Output

The Listener plug-in outputs to Listener.log the C code necessary to play an event, including the initialization of event parameters, descriptors, and references. The Listener plug-in records user-level events during a working session of Photoshop. The C code is organized into a series of function definitions, each of which represent a Photoshop event. The most recent event is appended to the previous event in Listener.log. Keep in mind that not all Photoshop user-level events are actionable. Therefore, not all events have corresponding function definitions in Listener.log.

PlayeventNotify Function

Observer the Listener plug-in output for yourself. First, you must launch Photoshop. Depending on which operating system you are running, you must go to the location of Listener.log and open it. You should see the PlayeventNotify function definition. The PlayeventNotify function should look similar to the following code:

```
SPErr PlayeventNotify(/*your parameters go here*/void)
{
     PIActionDescriptor result = NULL;
     DescriptorTypeID runtimeID = 0;
     DescriptorTypeID runtimeTypeID = 0;
     SPErr error = kSPNoError;
     // Move this to the top of the routine!
```

```
        PIActionDescriptor desc00000188 = NULL;
        error = sPSActionDescriptor->Make(&desc00000188);
        if (error) goto returnError;
        error = sPSActionDescriptor->PutEnumerated(desc00000188,
                                                   keyWhat,
                                                   typeNotify,
                                                   enumFirstIdle);
        if (error) goto returnError;
        error = sPSActionControl->Play(&result,
                                       eventNotify,
                                       desc00000188,
                                       plugInDialogSilent);
        if (error) goto returnError;
    returnError:
        if (result != NULL) sPSActionDescriptor->Free(result);
        if (desc00000188 != NULL) sPSActionDescriptor-
    >Free(desc00000188);
        return error;
    }
```

The PlayeventNotify function always occurs when Photoshop is launched. PlayeventNotify represents no useful code. Do not paste it into your source code.

Note that when you toggle between Photoshop and Listener.log, you may not see changes to the contents of Listener.log while the Listener plug-in is writing to Listener.log. You may need to refresh the Listener.log file by closing it and opening it again.

Understanding Listener Output

This section explains line by line four samples of function definitions created by the Listener plug-in. The samples are ordered by increasing difficulty. The functions include PlayeventPaste, PlayeventGaussianBlur, PlayeventMake, and PlayeventSet. There are four parts to the explanation of each function: 1) a description of the user-level event, 2) the function definition corresponding to the user-level event, 3) an overview of the function definition, and 4) a line by line description of the function definition.

Not all the variables in the sample code are explained. A list of variables, such as sPSActionDescriptor, sPSActionReference, sPSActionControl are defined in the Common Listener Variables section.

PlayeventPaste Function

PlayeventPaste function represents the user-level event of selecting from the menu "Edit" > "Paste" menuitem or Ctrl+V on the keyboard. The PlayeventPaste function pastes the data in the clipboard to the current layer. The Listener plug-in outputs C code to Listener.log similar to the following code:

```
    SPErr PlayeventPaste(/*your parameters go here*/void)
    {
        PIActionDescriptor result = NULL;
        DescriptorTypeID runtimeID = 0; //only used in getting strings
        DescriptorTypeID runtimeTypeID = 0; //only used in getting
    strings
        SPErr error = kSPNoError;

        error = sPSActionControl->Play(&result,
                                       eventPaste,
```

```
                                        NULL,
                                        plugInDialogSilent);
        if (error) goto returnError;
    returnError:
        if (result != NULL) sPSActionDescriptor->Free(result);
        return error;
    }
```

The PlayeventPaste function is the simplest type of function definition created by the Listener plug-in. Each function that the Listener plug-in creates provides "runtimeID" and "runtimeTypeID", which are values used in getting strings. These values can be omitted for the purpose of Playevent-Paste. The Play function executes the desired event and takes four parameters: the address of the result descriptor, the event name, the event descriptor, and the dialog mode. (what's the purpose of result?) Since there are no event parameters required in pasting, NULL is passed as the event descriptor. The dialog mode has three possible arguments: plugInDialog-DontDisplay, plugInDialogDisplay, and plugInDialogSilent. The most common mode, plugInDialogSilent never displays dialogs, therefore, allowing for complete automation with no user interaction. The plugInDialogDisplay mode displays the related dialog box for every step of the event. The plugIn-DialogDontDisplay mode displays a dialog box only if the function comes across invalid data for execution.

Throughout the function, each function call returns an error value. If an error occurs at any point, descriptors are freed, the error is returned, and the function ends.

You must edit the Listener plug-in output to suit your purpose. The PlayeventPaste function may be further simplified by removing "runtimeID" and "runtimeTypeID". The new version of PlayeventPaste is reformatted for readability and renamed to Paste. Since "eventPaste" requires no parameters, there's no need to pass any parameters into the function.

```
SPErr Paste(void)
{
      PIActionDescriptor result = NULL;
      SPErr error = kSPNoError;

      error = sPSActionControl->Play(&result,
                                     eventPaste,
                                     NULL,
                                     plugInDialogSilent);
      if (error) goto returnError;
      returnError:
      if (result != NULL) sPSActionDescriptor->Free(result);
      return error;
}
```

PlayeventGaussianBlur Function

The PlayeventGaussianBlur function corresponds to the user-level event of selecting from the menu "Filter" > "Blur" > "Gaussian Blur...". The PlayeventGaussianBlur function applies the Gaussian Blur filter to the current layer. The Listener plug-in outputs C code to Listener.log similar to the following code:

```
SPErr PlayeventGaussianBlur(/*your parameters go here*/void)
{
      PIActionDescriptor result = NULL;
      DescriptorTypeID runtimeID = 0;
      DescriptorTypeID runtimeTypeID = 0;
      SPErr error = kSPNoError;
```

```
        // Move this to the top of the routine!
        PIActionDescriptor desc00000050 = NULL;
        error = sPSActionDescriptor->Make(&desc00000050);
        if (error) goto returnError;
        error = sPSActionDescriptor->PutFloat(desc00000050,
                                              keyRadius,
                                              2);
        if (error) goto returnError;
        error = sPSActionControl->Play(&result,
                                       eventGaussianBlur,
                                       desc00000050,
                                       plugInDialogSilent);
        if (error) goto returnError;
    returnError:
        if (result != NULL) sPSActionDescriptor->Free(result);
        if (desc00000050 != NULL) sPSActionDescriptor-
    >Free(desc00000050);
        return error;
    }
```

Unlike PlayeventPaste, the PlayeventGaussianBlur function requires an event descriptor, named "desc00000050". The Make function dynamically allocates memory for "descriptor". The PutFloat function puts the keyRadius value of 2 into "descriptor". There are several types of 'Put' functions that are used to put values into descriptors. All 'Put' functions are listed in the PITerminology.h file of the SDK. Since PlayeventGaussianBlur requires an event descriptor, "desc00000050", instead of NULL, is passed into the Play function.

The Listener plug-in does not do everything perfectly. Notice the line after the comment, "// Move this to the top of the routine!". You must move the initialization of descriptors to the beginning of the function. Otherwise, the error checking is incorrect.

Below is a simplified, cleaner version of PlayeventGaussianBlur. Notice that the function name and the descriptor names are different. Plus, the radius of Gaussian Blur is passed into the function.

```
    SPErr GaussianBlur(double radius)
    {
          PIActionDescriptor descriptor = NULL;
          PIActionDescriptor result = NULL;
          SPErr error = kSPNoError;

          error = sPSActionDescriptor->Make(&descriptor);
          if (error) goto returnError;
          error = sPSActionDescriptor->PutFloat(descriptor,
                                                keyRadius,
                                                radius);
          if (error) goto returnError;
          error = sPSActionControl->Play(&result,
                                         eventGaussianBlur,
                                         descriptor,
                                         plugInDialogSilent);
          if (error) goto returnError;
          returnError:
          if (result != NULL) sPSActionDescriptor->Free(result);
          if (descriptor != NULL) sPSActionDescriptor->Free(descriptor);
          return error;
    }
```

PlayeventMake Function

The PlayeventMake function represents the user-level event of selecting from the menu "Layer" > "New" > "Layer" and clicking OK or Shift+Ctrl+N on the keyboard. The function creates a new layer. The Listener plug-in outputs C code to Listener.log similar to the following code:

```
SPErr PlayeventMake(/*your parameters go here*/void)
{
     PIActionDescriptor result = NULL;
     DescriptorTypeID runtimeID = 0;
     DescriptorTypeID runtimeTypeID = 0;
     SPErr error = kSPNoError;

     // Move this to the top of the routine!
     PIActionDescriptor desc00000058 = NULL;
     error = sPSActionDescriptor->Make(&desc00000058);
     if (error) goto returnError;
          // Move this to the top of the routine!
          PIActionReference ref00000030 = NULL;
          error = sPSActionReference->Make(&ref00000030);
          if (error) goto returnError;
          error = sPSActionReference->PutClass(ref00000030,
                                               classLayer);
          if (error) goto returnError;
     error = sPSActionDescriptor->PutReference(desc00000058,
                                               keyNull,
                                               ref00000030);
     if (error) goto returnError;
     error = sPSActionControl->Play(&result,
                                    eventMake,
                                    desc00000058,
                                    plugInDialogSilent);
     if (error) goto returnError;
returnError:
     if (result != NULL) sPSActionDescriptor->Free(result);
     if (desc00000058 != NULL) sPSActionDescriptor-
>Free(desc00000058);
     if (ref00000030 != NULL) sPSActionReference->Free(ref00000030);
     return error;
}
```

The PlayeventMake function contains all the elements of the PlayeventGaussianBlur function. In addition, the function makes a reference, named "ref00000030". The PutClass function sets "ref00000030" to "classLayer". Subsequently, the PutReference function puts "reference" into the event descriptor, "desc00000058".

A simpler and cleaner version of PlayeventMake is renamed "MakeLayer". "desc00000058" and "ref00000030" are renamed and their initializations are relocated to the beginning of the function. No parameters need to be passed into the function.

```
SPErr MakeLayer(void)
{
      PIActionDescriptor descriptor = NULL;
      PIActionReference reference = NULL;
      PIActionDescriptor result = NULL;
      SPErr error = kSPNoError;

      error = sPSActionDescriptor->Make(&descriptor);
      if (error) goto returnError;
          // Move this to the top of the routine!
          error = sPSActionReference->Make(&reference);
          if (error) goto returnError;
```

```
            error = sPSActionReference->PutClass(reference,
                                                 classLayer);
            if (error) goto returnError;
        error = sPSActionDescriptor->PutReference(descriptor,
                                                 keyNull,
                                                 reference);
        if (error) goto returnError;
        error = sPSActionControl->Play(&result,
                                       eventMake,
                                       descriptor,
                                       plugInDialogSilent);
        if (error) goto returnError;
    returnError:
        if (result != NULL) sPSActionDescriptor->Free(result);
        if (descriptor != NULL) sPSActionDescriptor->Free(descriptor);
        if (reference != NULL) sPSActionReference->Free(reference);
        return error;
    }
```

PlayeventSet Function

The PlayeventSet function corresponds to the user-level event of creating a selection using the Rectangular Marquee Tool. The Listener plug-in outputs C code to Listener.log similar to the following code:

```
    SPErr PlayeventSet(/*your parameters go here*/void)
    {
        PIActionDescriptor result = NULL;
        DescriptorTypeID runtimeID = 0;
        DescriptorTypeID runtimeTypeID = 0;
        SPErr error = kSPNoError;

        // Move this to the top of the routine!
        PIActionDescriptor desc00000068 = NULL;
        error = sPSActionDescriptor->Make(&desc00000068);
        if (error) goto returnError;
            // Move this to the top of the routine!
            PIActionReference ref00000018 = NULL;
            error = sPSActionReference->Make(&ref00000018);
            if (error) goto returnError;
            error = sPSActionReference->PutProperty(ref00000018,
                                                    classChannel,
                                                    keySelection);
            if (error) goto returnError;
        error = sPSActionDescriptor->PutReference(desc00000068,
                                                 keyNull,
                                                 ref00000018);
        if (error) goto returnError;
            // Move this to the top of the routine!
            PIActionDescriptor desc00000070 = NULL;
            error = sPSActionDescriptor->Make(&desc00000070);
            if (error) goto returnError;
            error = sPSActionDescriptor->PutUnitFloat(desc00000070,
                                                      keyTop,
                                                      unitDistance,
                                                      63);
            if (error) goto returnError;
            error = sPSActionDescriptor->PutUnitFloat(desc00000070,
                                                      keyLeft,
                                                      unitDistance,
```

```
                                                        34);
            if (error) goto returnError;
            error = sPSActionDescriptor->PutUnitFloat(desc00000070,
                                                        keyBottom,
                                                        unitDistance,
                                                        296);
            if (error) goto returnError;
            error = sPSActionDescriptor->PutUnitFloat(desc00000070,
                                                        keyRight,
                                                        unitDistance,
                                                        245);
            if (error) goto returnError;
        error = sPSActionDescriptor->PutObject(desc00000068,
                                                keyTo,
                                                classRectangle,
                                                desc00000070);
        if (error) goto returnError;
        error = sPSActionControl->Play(&result,
                                    eventSet,
                                    desc00000068,
                                    plugInDialogSilent);
        if (error) goto returnError;
    returnError:
        if (result != NULL) sPSActionDescriptor->Free(result);
        if (desc00000068 != NULL) sPSActionDescriptor-
    >Free(desc00000068);
        if (ref00000018 != NULL) sPSActionReference->Free(ref00000018);
        if (desc00000070 != NULL) sPSActionDescriptor-
    >Free(desc00000070);
        return error;
    }
```

In contrast to the previous functions, PlayeventSet involves an event parameter descriptor - a descriptor that 'describes' an event parameter. This parameter is more complex than keyRadius of the eventGaussianBlur. "desc00000070" describes the rectangle by holding the values that represent the four corners of the rectangle. The PutUnitFloat function, instead of PutFloat function, is used to put the values of keyTop, keyLeft, keyBottom, and keyRight into "desc00000070". Subsequently, "desc00000070" is put into the event descriptor, "desc00000068" using the PutObject function. Renaming descriptor variables in the PlayeventSet function would make the function more readable and meaningful. The RectangleSelection function is a cleaner version of the PlayeventSet function. The function takes four parameters ("top", "left", "bottom", "right") for the corners of the rectangle. The descriptors, "desc00000068" and "desc00000070" were changed to "descriptor" and "rectDescriptor". As usual, you must move the initializations of the descriptors to the beginning of the function and remove the variables, "runtimeID" and "runtimeTypeID".

```
    SPErr RectangleSelection(double top,
                             double left,
                             double bottom,
                             double right)
    {
          SPErr error = kSPNoError;
          PIActionDescriptor result;
          PIActionDescriptor descriptor;
          PIActionReference reference;
          PIActionDescriptor rectDescriptor;

          error = sPSActionDescriptor->Make(&descriptor);
          if (error) goto returnError;
          error = sPSActionReference->Make(&reference);
```

```
        if (error) goto returnError;
        error = sPSActionDescriptor->Make(&rectDescriptor);
        if (error) goto returnError;
        error = sPSActionReference->PutProperty(reference,
                                                classChannel,
                                                keySelection);
        if (error) goto returnError;
        error = sPSActionDescriptor->PutReference(descriptor,
                                                  keyNull,
                                                  reference);
        if (error) goto returnError;
        error = sPSActionDescriptor->PutUnitFloat(rectDescriptor,
                                                  keyTop,
                                                  unitDistance,
                                                  top);
        if (error) goto returnError;
        error = sPSActionDescriptor->PutUnitFloat(rectDescriptor,
                                                  keyLeft,
                                                  unitDistance,
                                                  left);
        if (error) goto returnError;
        error = sPSActionDescriptor->PutUnitFloat(rectDescriptor,
                                                  keyBottom,
                                                  unitDistance,
                                                  bottom);
        if (error) goto returnError;
        error = sPSActionDescriptor->PutUnitFloat(rectDescriptor,
                                                  keyRight,
                                                  unitDistance,
                                                  right);
        if (error) goto returnError;
        error = sPSActionDescriptor->PutObject(descriptor,
                                               keyTo,
                                               classRectangle,
                                               rectDescriptor);
        if (error) goto returnError;
        error = sPSActionControl->Play(&result,
                                       eventSet,
                                       descriptor,
                                       plugInDialogSilent);
        if (error) goto returnError;

    returnError:
        /*free descriptors and references*/
        if (result != NULL) sPSActionDescriptor->Free(result);
        if (descriptor != NULL)sPSActionDescriptor->Free(descriptor);
        if (reference != NULL)sPSActionReference->Free(reference);
        if (rectDescriptor != NULL) sPSActionDescriptor->Free(rectDe-
scriptor);
        return error;
}/*end RectangleSelection*/
```

Common Listener Variables

The Listener plug-in consistently uses the following data types and pointer variable names in its output. The MakeSelectFill sample plug-in uses all the common Listener variables.

PIActionDescriptor
the data type for descriptors, including result descriptors, event descriptors, and event parameter descriptors

PIActionReference
the data type for references

sPSActionDescriptor
a pointer to the PSActionDescriptorProcs suite containing the Make function and various 'Put' functions

sPSActionReference
a pointer to the PSActionReferenceProcs suite containing the Make function and PutProperty function

sPSActionControl
a pointer to the PSActionControlProcs suite containing the Play function

Conclusion

After reading this chapter, you should be able to build the Listener plug-in successfully and have an in depth understanding of the functions that the Listener plug-in creates. You are ready to incorporate the functions into an project workspace. The project workspace chapter shows you how to create a Photoshop project workspace from scratch and describes the proper settings for an automation plug-in. If you do not want to create a project workspace from scratch, you may use the Template sample plug-in as a starting point. The MakeSelectFill sample plug-in is an example of incorporating functions created by the Listener plug-in to build an automation plug-in. The MakeSelectFill plug-in uses the MakeDoc and RectangleSelection functions that were mentioned in this chapter.

4 Project Workspace

Introduction

This chapter gives you step-by-step instructions to set up a project workspace for a Photoshop automation plug-in. You should already know how to run the Listener plug-in and create functions using the Listener plug-in. If you do not have a thorough understanding of the Listener plug-in, you must read the Listener chapter. The majority of this chapter explains the project set-up procedure for Metroworks CodeWarrior and for Microsoft Visual C++. This project set-up procedure covers creating the appropriate directory structure for the project, creating a new project workspace, entering the proper project settings, adding the necessary source and resource files, building the project, creating a plug-in shortcut/alias, and successfully loading the plug-in into Photoshop.
The final product of the project set-up procedure resembles the Template project. After completing this chapter, you may verify that you have built a project workspace correctly by comparing your's with that of the Template sample plug-in.
Additionally, the MakeSelectFill sample plug-in project workspace demonstrates incorporating functions which call Photoshop events. In contrast to the Template plug-in, the MakeSelectFill plug-in includes MakeDoc, RectangleSelection, and ColorFill functions, all of which were created by the Listener plug-in.

Creating the Directory Structure

The project workspace that is discussed in this chapter follows a directory structure in which your plug-in folder is contained in the Automation folder of the SDK. The plug-in name for this project is "Auto". You may replace it with your plug-in's name. You must create three folders: "Auto", "Common", and "Win" or "Mac". You need to create the folders having the following paths:

For Mac
MacHD:Adobe Photoshop 6.0 SDK:Sample Code:Automation:Auto
MacHD:Adobe Photoshop 6.0 SDK:Sample Code:Automation:Auto:Mac
MacHD:Adobe Photoshop 6.0 SDK:Sample Code:Automation:Auto:Common

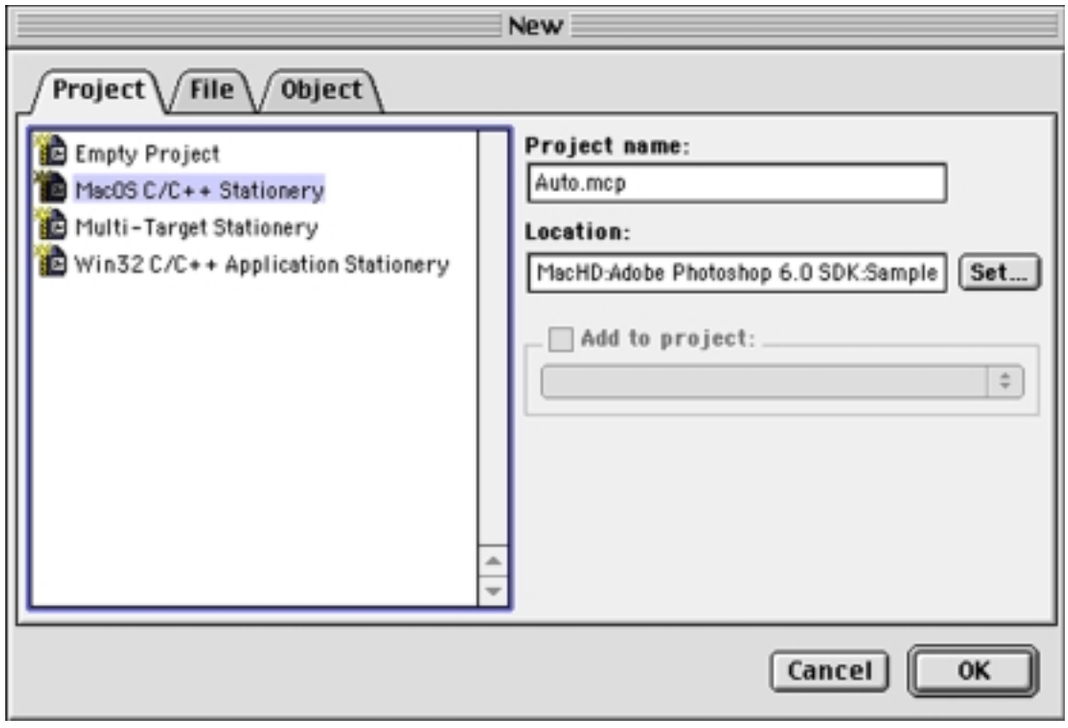
For Windows
D:\Adobe Photoshop 6.0 SDK\SampleCode\Automation\Auto
D:\Adobe Photoshop 6.0 SDK\SampleCode\Automation\Auto\Win
D:\Adobe Photoshop 6.0 SDK\SampleCode\Automation\Auto\Common

Metrowerks CodeWarrior Project, Pro 5

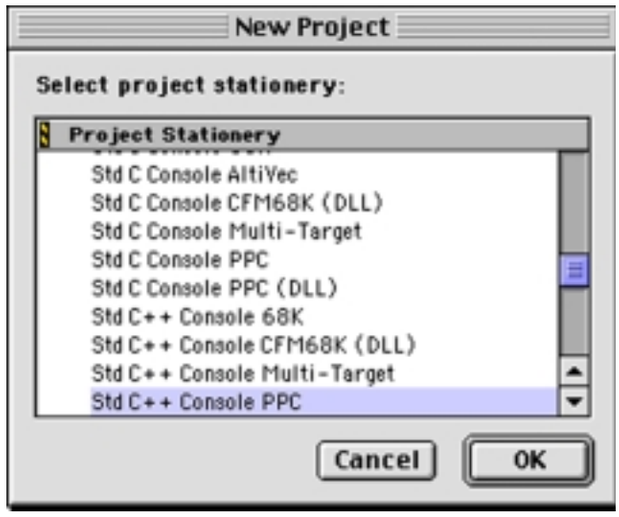
Creating A Project Workspace

All Macintosh Photoshop plug-ins have three basic properties. They are shared libraries, they have a PiPL resource, and they have a program entry-point
or main function.

1. Launch Metrowerks CodeWarrior Project, Pro 5
2. Choose from the menu "File" > "New".
3. From the "New" dialog, click on the "Project" tab.
4. Select "MacOS C/C++ Stationery"
5. In the "Project name:" field, enter "Auto.mcp".
6. In the "Location:" field, enter the path to your project, similar to the following directory:
7. "MacHD:Adobe Photoshop 6.0 SDK:Sample Code:Automation:Auto:Mac"
8. Ensure that "Add to project:" is NOT selected.
9. Click "OK".



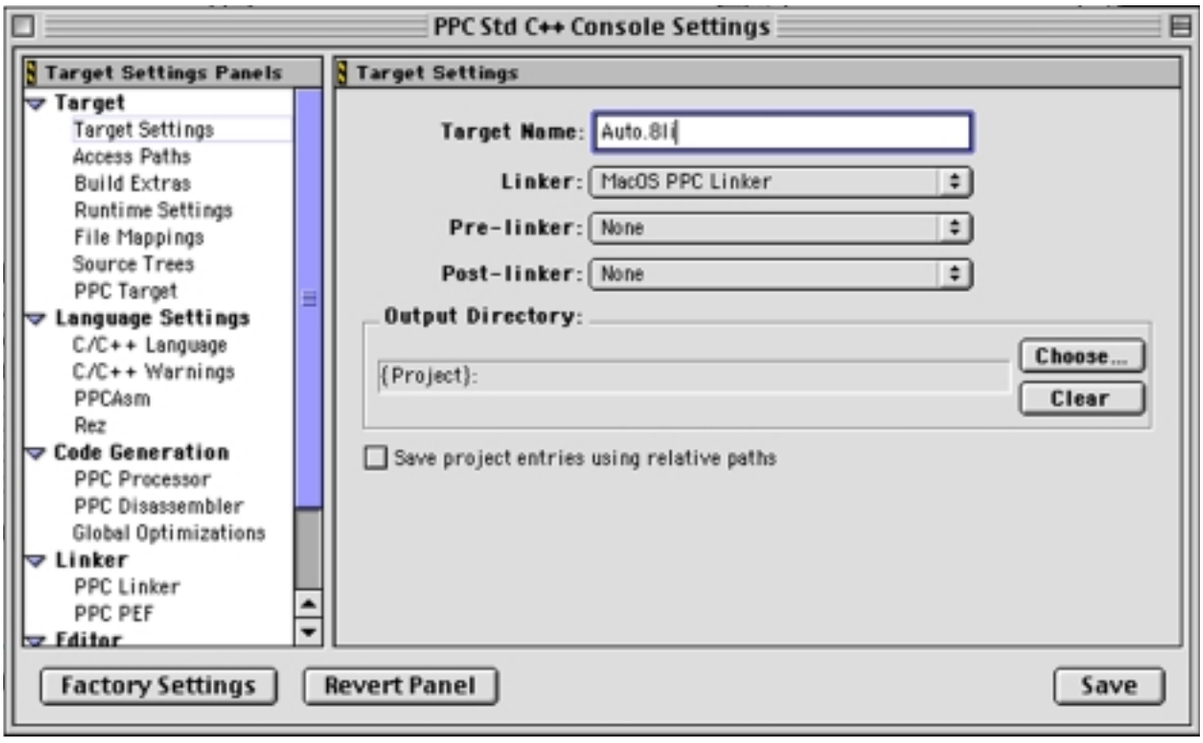
10. From the "New Project" dialog box, click on the triangle next to "Standard Console".
11. Select "Std C++ Console PPC".
12. Click "OK".



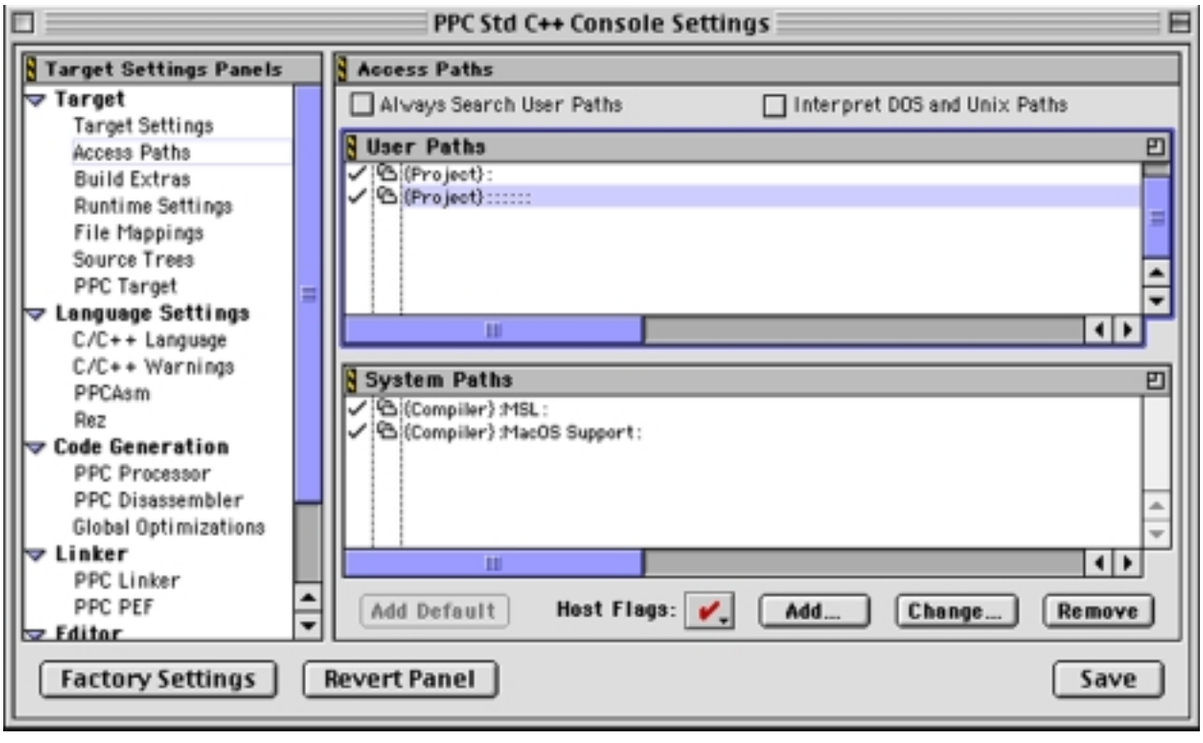
Entering Project Settings

Now that you have a workspace, you need to fix up the properties to build a proper Photoshop plug-in.

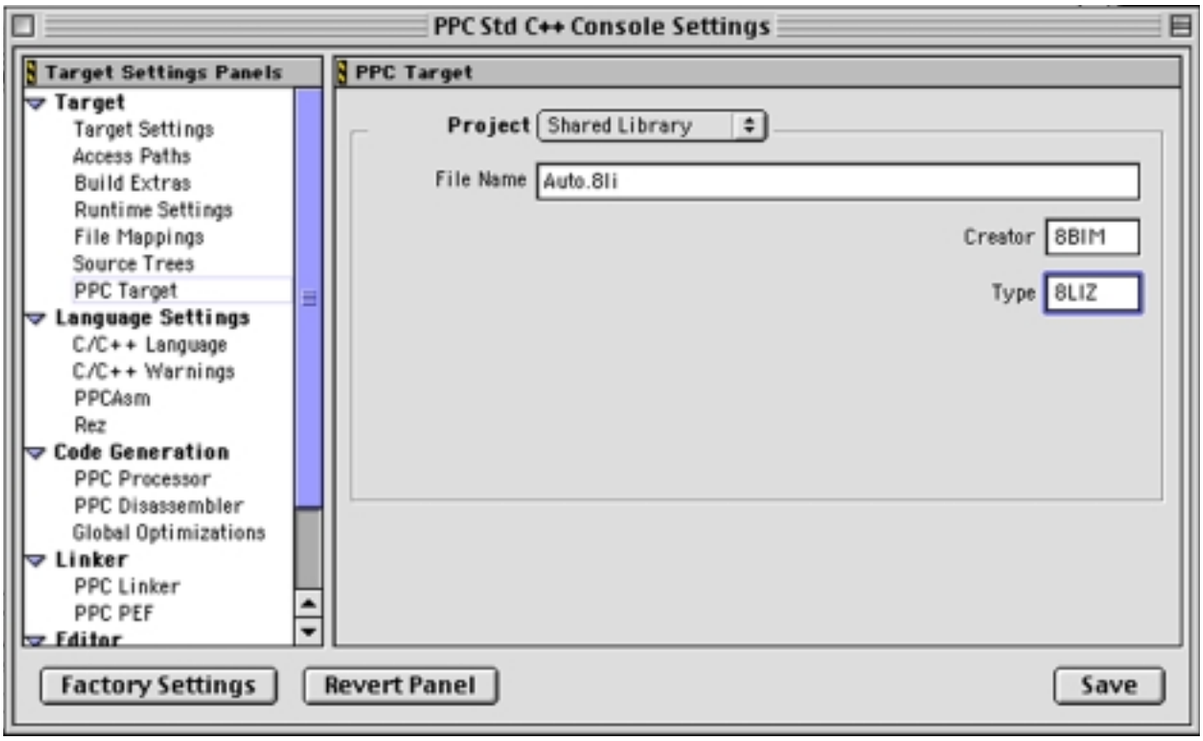
1. Choose from the menu "Edit" > "PPC Std C++ Console Settings..."
2. In the left window of the "PPC Std C++ Console Settings" dialog, click on "Target Settings".
3. In the "Target Name:" field, enter "Auto.8li".



4. In the left window of the dialog, click "Access Paths".
5. Under the "User Paths:" panel, click on "{Project}:".
6. Click the "Add" button and navigate to and click on the "Adobe Photoshop 6.0 SDK" folder.
7. From the "Relative To:" pop-up, choose "Project".
8. Click the "Choose" button.



9. In the left window of the dialog, click on "PPC Target".
10. Change the "Project" setting to "Shared Library".
11. Change the "File Name" entry to "Auto.8li"
12. Change the "Creator" entry to "8BIM".
13. Change the "Type" entry to "8LIZ".

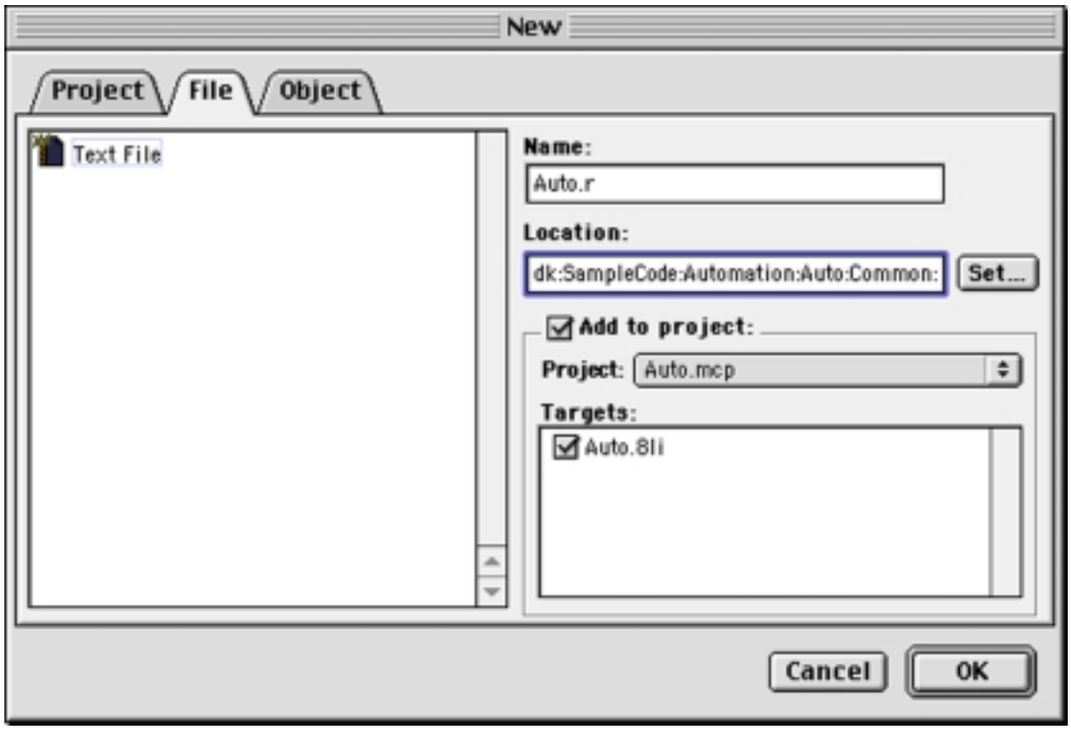


14. In the left window of the dialog, click on "C/C++ Language".
15. Disable the "ANSI Strict" checkbox.
16. In the left window of the dialog, click on "Rez".
17. Change the "Prefix File" entry to "PIRezMac.h"
18. In the left window of the dialog, click on "PPC Processor".
19. Change the "Struct Alignment:" setting to "68K".
20. In the left window of the dialog, click on "PPC Linker".
21. Change the "Main:" entry to "PluginMain".
22. Click "Save" and close the dialog.

Adding A Resource File

You need a resource file called "Auto.r".

1. Back in CodeWarrior, choose from the menu "File" > "New".
2. Click on "File" tab.
3. Select "Text File".
4. In the "Name:" field, enter "Auto.r".
5. In the "Location:" field, enter the path similar to the following directory:
6. "MacHD:Adobe Photoshop 6.0 SDK:Sample Code:Automation:Auto:Common"
7. Click "Save".
8. Check the "Add to project:" box.
9. Click "OK".

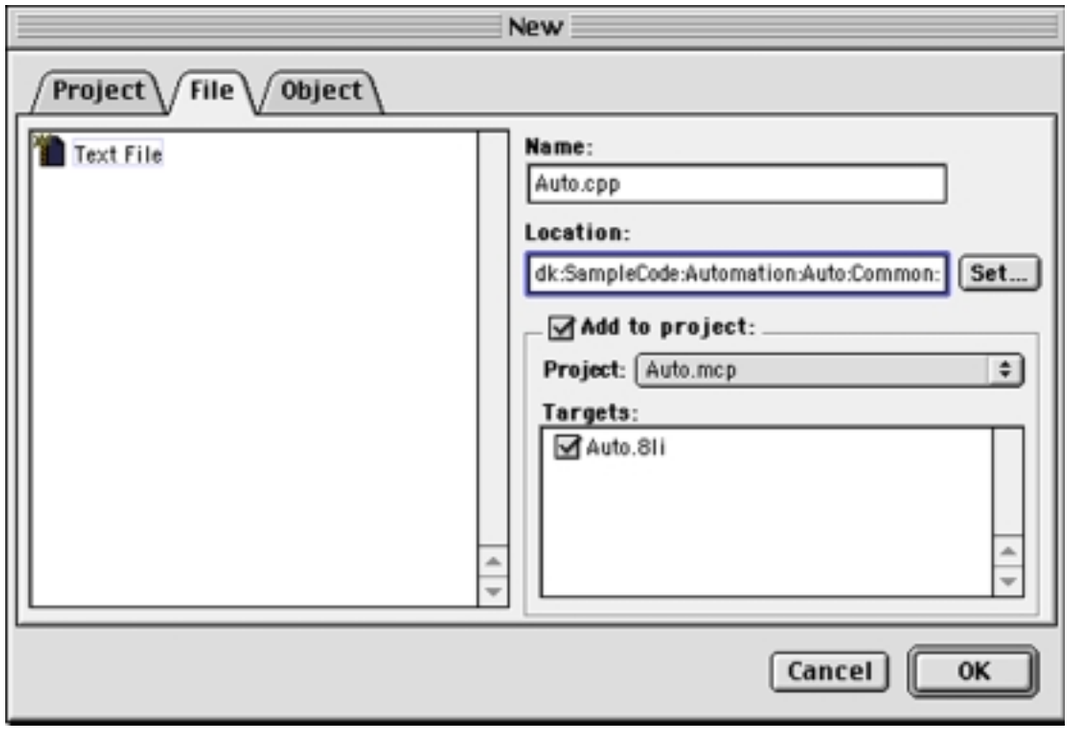


10. The "Auto.r file" that you just created opens up.
11. Paste the contents of "Auto.r", located at the end of this section.
12. Save and close the "Auto.r" file.

Adding The Main Source File

CodeWarrior automatically creates a HelloWorld.cp file in your project. You need to add the PiPL resource to the project, so that Photoshop recognizes your project output as a Photoshop automation plug-in.

1. From the "Auto.mcp" project under the "Sources" folder, click on "HelloWorld.cp" and command-delete "HelloWorld.cp" from the project.
2. In your project, click on the "Sources" folder.
3. Choose from the menu "File" > "New".
4. Click on the "File" tab.
5. Select "Text File".
6. In the "Name:" field, enter "Auto.cpp".
7. Ensure that the "Add to project:" option is selected.
8. In the "Location:" field, set the path similar to the following directory:
9. "MacHD:Adobe Photoshop 6.0 SDK:Sample Code:Automation:Auto:Common"
10. Click "Save".
11. Set the "Project:" to "Auto.mcp".
12. Click "OK".
13. The "Auto.cpp" file that you just created opens up.
14. Paste the contents of "Auto.cpp", located at the end of this section.
15. Save and close "Auto.cpp" file.



Adding Other Necessary Source Files

1. Back in Auto.mcp, select the "Sources" folder.
2. Choose from the menu "Project" > "Add Files".
3. Browse to the following directory of the SDK:
4. "MacHD:Adobe Photoshop 6.0 SDK:Sample Code:Common:Sources"
5. Double click "PIUGet.cpp".
6. Repeat the previous steps to add "PIUNew.cpp".
7. Repeat the previous steps to add "PIUSuites.cpp".

Making The Project

Now you have all the code necessary to create an automation plug-in. Your project settings complete. Your project is ready to be built.

1. Choose from the menu "Project" > "Make".
2. If your build is successful, "Auto.8li" file should appear in the Mac folder.
3. You should have no errors or warnings after the build. If you encounter any, go over the previous steps. Make sure there are no misspellings.

Creating A Plug-In Alias To Plug-Ins Folder

You need to get "Auto.8li" into the Photoshop "Plug-Ins" folder. That way, Photoshop loads the plug-in during the next launch of Photoshop.

1. Find the "Auto.8li" file that you just created.
2. Select the "Auto.8li" file.
3. Create an alias to the folder, by choosing from the menu "File" > "Make Alias"
4. Drag the alias to the "Plug-Ins" folder of Photoshop.

Loading the Plug-In Into Photoshop

1. Launch Photoshop.
2. Create a new document.
3. To test whether Auto is working, choose under the menu Apple Icon > "About Plug-in" > "Auto". Verify that the About feature works properly.
4. Verify the functionality of Auto by choosing from the menu "File" > "Automate" > "Auto".

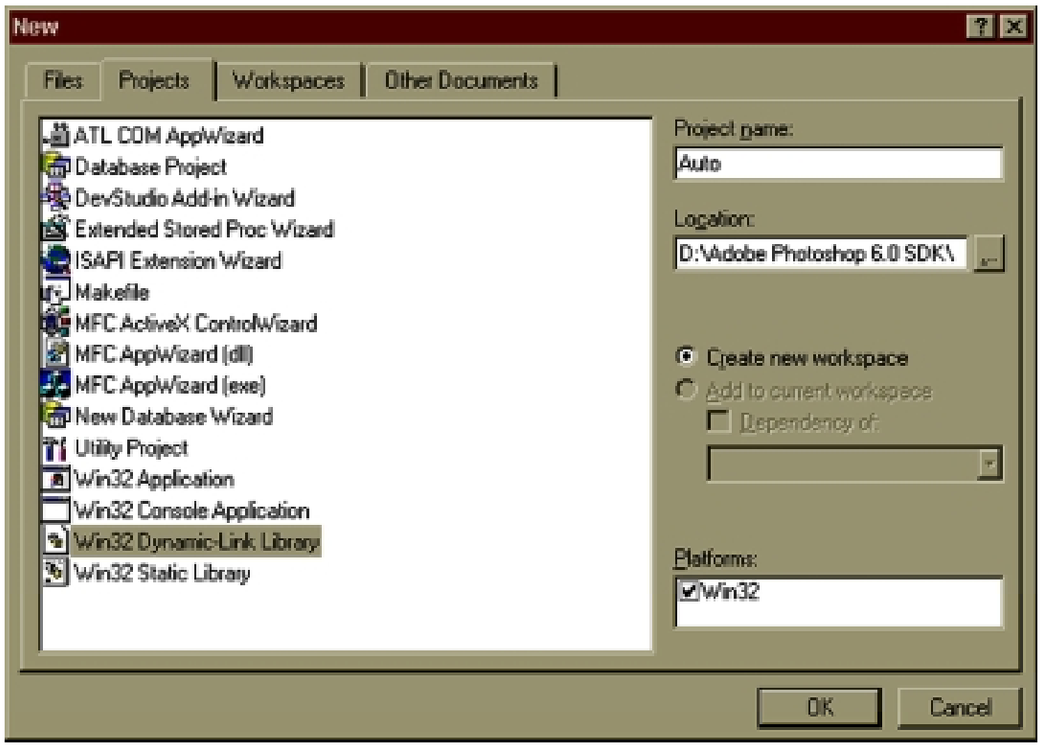
Congratulations!! Now, you have a functional Photoshop automation plug-in.

Microsoft Visual C++, Version 6.0

Creating A Project Workspace

All Windows Photoshop plug-ins have three basic properties. They are shared libraries, they have a PiPL resource, and they have a program entry-point
or main function.

1. Launch Microsoft Visual C++, Version 6.0.
2. Choose from the menu "File" > "New".
3. Click on the "Projects" tab.
4. From the list of projects, choose "Win32 Dynamic-Link Library".
5. In the "Project name:" field, type "Auto" (or the name of your plug-in).
6. In the "Location:" field, type in the location for your project. Set the path similar to the following:
7. "D:\Adobe Photoshop 6.0 SDK\SampleCode\Automation\Auto\Win".
8. Ensure the "Create new workspace" is selected and "Platforms Win32" is checked.
9. Click "OK".



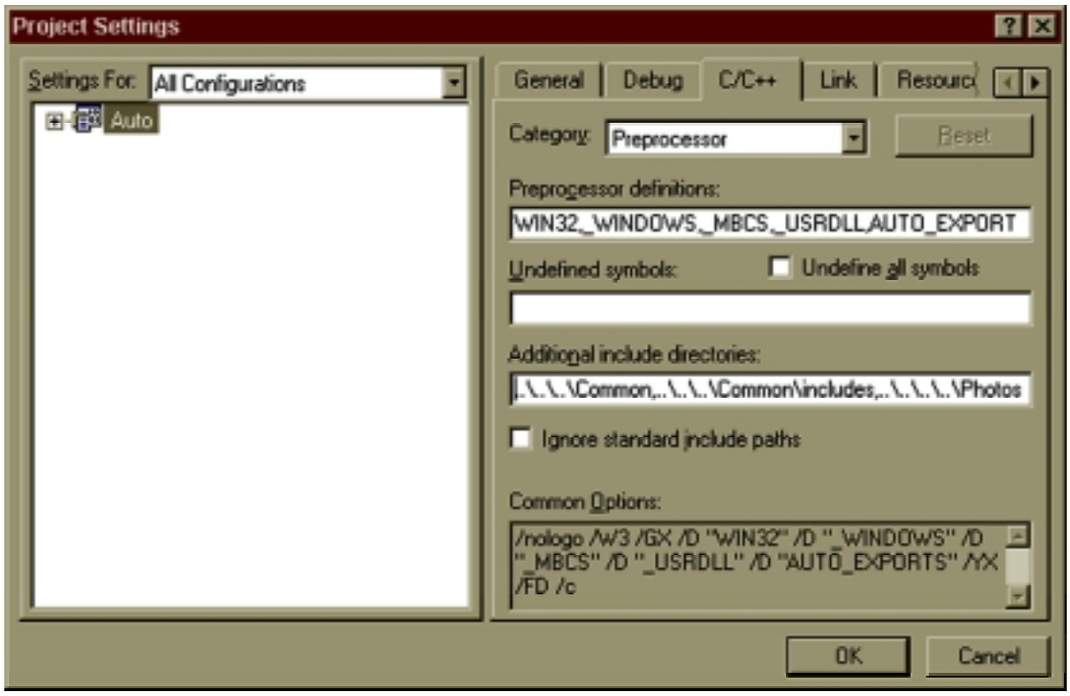
10. The "Win32 Dynamic-Link Library" dialog appears.
11. Ensure that "An empty DLL project." is selected.
12. Click "Finish".
13. The "New Project Information" dialog appears.
14. Click "OK".

Entering Project Settings

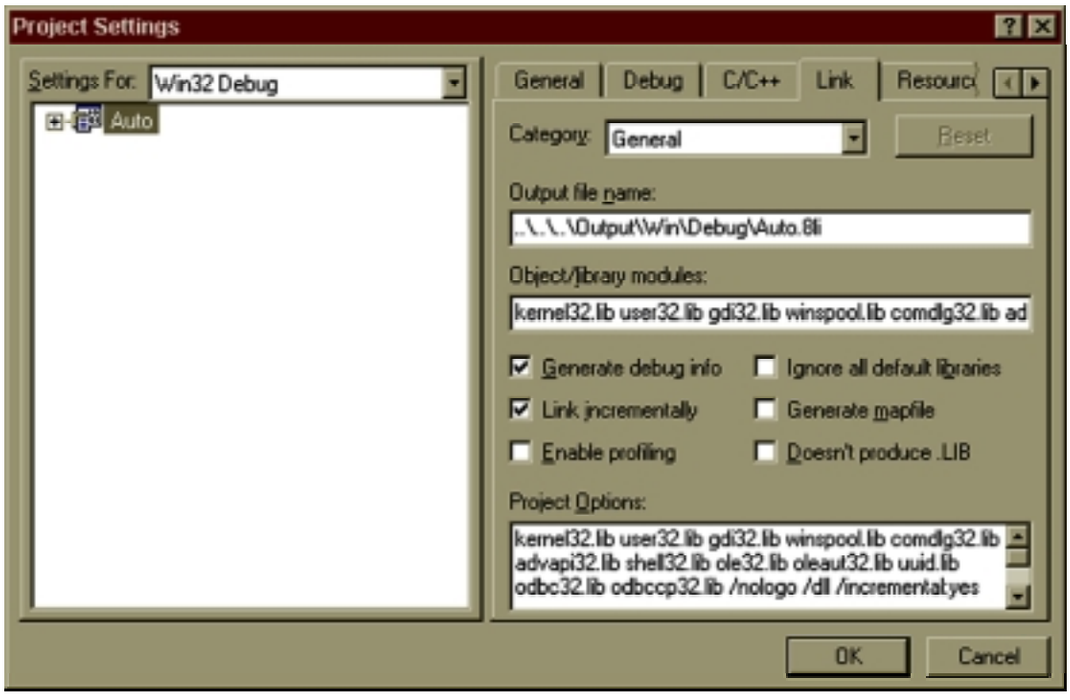
At this point, you have a workspace. You must fix the project settings to build a proper Photoshop plug-in.

1. Choose from the menu "Project" > "Settings".
2. Click on "General" tab.
3. In the "Settings for:" field, select "All Configurations".
4. Under both the "Intermediate files:" and "Output files:" fields, type "temp".
5. Click on "C/C++" tab.
6. Set the "Category" to "Preprocessor".
7. Enter the following paths into the "Additional include directories:".
8. NOTE: If you did not put your plug-in into the SDK, these relative paths will not work. Make sure your paths point to these folders that are in the Photoshop SDK.
   "..\..\..\Common,..\..\..\Common\includes,..\..\..\..\PhotoshopAPI,..\..\..\..\PhotoshopAPI\ADM

,..\..\..\..\PhotoshopAPI\Photoshop,..\..\..\..\PhotoshopAPI\General,..\..\..\..\Photosho-pAPI\PICA_SP"



9. Click on the "Link" tab.
10. Under the "Category" field, select "General".
11. Click on "Settings for:" field. Select "Win32 Release".
12. Change the "Output file name:" to (replace the word, "Auto" with your plug-in's name)
13. "..\..\..\Output\Win\Release\Auto.8li"
14. Click on "Settings for:" field. Select "Win32 Debug".
15. Change the "Output file name:" to (replace the word, "Auto" with your plug-in's name)
16. "..\..\..\Output\Win\Debug\Auto.8li"
17. Click OK.

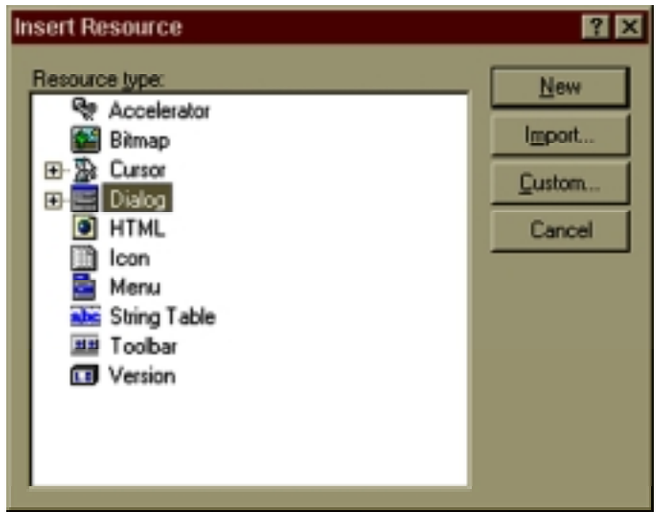


Adding A Resource File

Now you have a blank project to work from. You must add a resource file which contains the PiPL resource.

1. Choose from the menu "File" > "New".
2. Click on "Files" tab.
3. From the list of file types, choose "Resource Script".

4. In the "File name:" field, type "Auto.rc".
5. In the "Location:" field, type in the location for your project. Set the path similar to the following:
6. "D:\Adobe Photoshop 6.0 SDK\SampleCode\Automation\Auto\Win".
7. Ensure that the "Add to project" option is NOT checked.
8. Click OK.
9. Right click on the "Auto.rc" file (folder icon) and select "Insert".
10. From the list of resource types, select "Dialog".
11. Click "New".



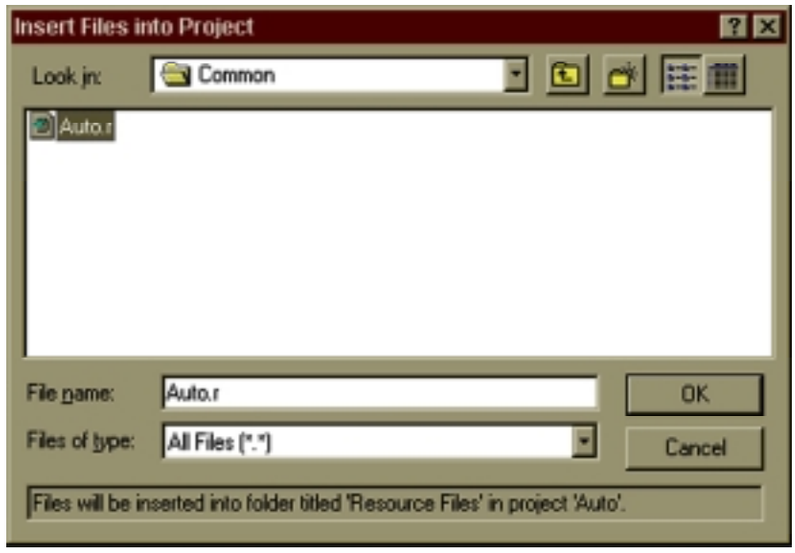
12. Close the "Auto.rc" file in the upper-right window.
13. Close the resource editor by closing the upper-right window.
14. A dialog pops up asking if you want to save changes.
15. Click "Yes".
16. Choose from the menu "File" > "Open".
17. Select (don't open) "Auto.rc" that you just created, located in the "Win" folder.
18. In the "Open as:" field, select "Text".
19. Click "Open".
20. On approximately line 43, replace the line: "\r\n" with the line:
21. ""#include ""temp\\Auto.pipl""\r\n""
22. Scroll down to the end of the file to approximately line 93.
23. Insert the following text between line containing "#ifndef APSTUDIO_INVOKED" and line containing "#endif" with the line: "#include "temp\\Auto.pipl""
24. Change all occurrences of "Auto" with your plug-in's name.
25. Save and close "Auto.rc".
26. In the "FileView" window, right click on "Resource Files".
27. Select "Add Files to Folder...".
28. Browse to and double click on the "Auto.rc" file that you just created.

Adding Necessary Resource File

This is the unique step of building a project on the Windows platform: the custom build step. You need to convert a Macintosh resource file into a Windows-compatible resource. you do this using the Visual C++ custom build step and the CNVTPIPL.EXE found in the SDK.

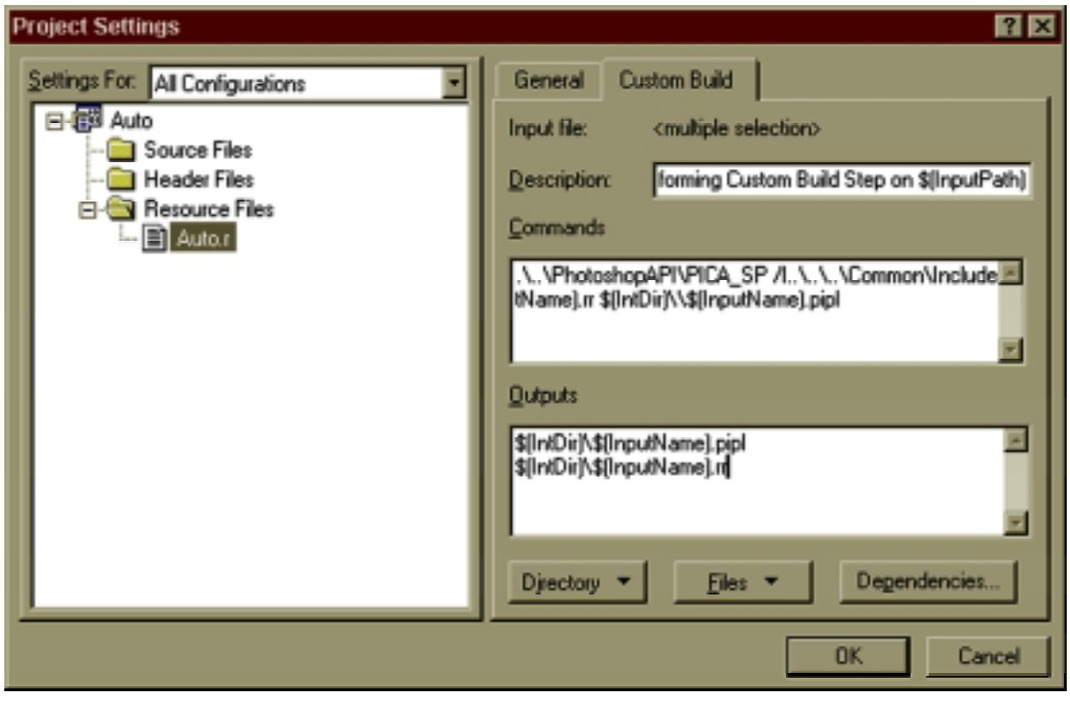
1. Choose from the menu "File" > "New".
2. Select "Files" tab and choose "Text File".
3. Ensure that "Add to project:" is NOT checked.
4. Type in "Auto.r" into the "File name:" field.
5. In the "Location:" field, type in the path location similar to the following:
6. "D:\Adobe Photoshop 6.0 SDK\SampleCode\Automation\Auto\Common"
7. Click "OK".
8. In the Auto.r file, paste the contents of Auto.r located at the end of this section.
9. Ensure that you replace "Auto" with the name of your plug-in.
10. Save and close "Auto.r".
11. In the "FileView" window, right click on "Resource Files".

12. Select "Add Files to Folder...".
13. Browse to and double click on the "Auto.r" file that you just created.



Entering Custom Build Settings

1. In the "FileView" window, right click on the "Auto.r" file .
2. Select "Settings".
3. Under "Settings for:", select "All Configurations".
4. Click on "Custom Build" tab and enter the following in the "Commands" section: (NOTE: You need to change the paths if you placed your project in a different folder.)
5. cl /I..\..\..\..\PhotoshopAPI\Photoshop /I..\..\..\..\PhotoshopAPI\PICA_SP /I..\..\..\Common\Includes /DMSWindows=1 /EP /Tc$(InputPath) > $(IntDir)\\$(InputName).rr
6. ..\..\..\Resources\Cnvtpipl.exe $(IntDir)\$(InputName).rr $(IntDir)\\$(InputName).pipl
7. In the "Outputs" section paste the following:
8. $(IntDir)\$(InputName).pipl
9. $(IntDir)\$(InputName).rr

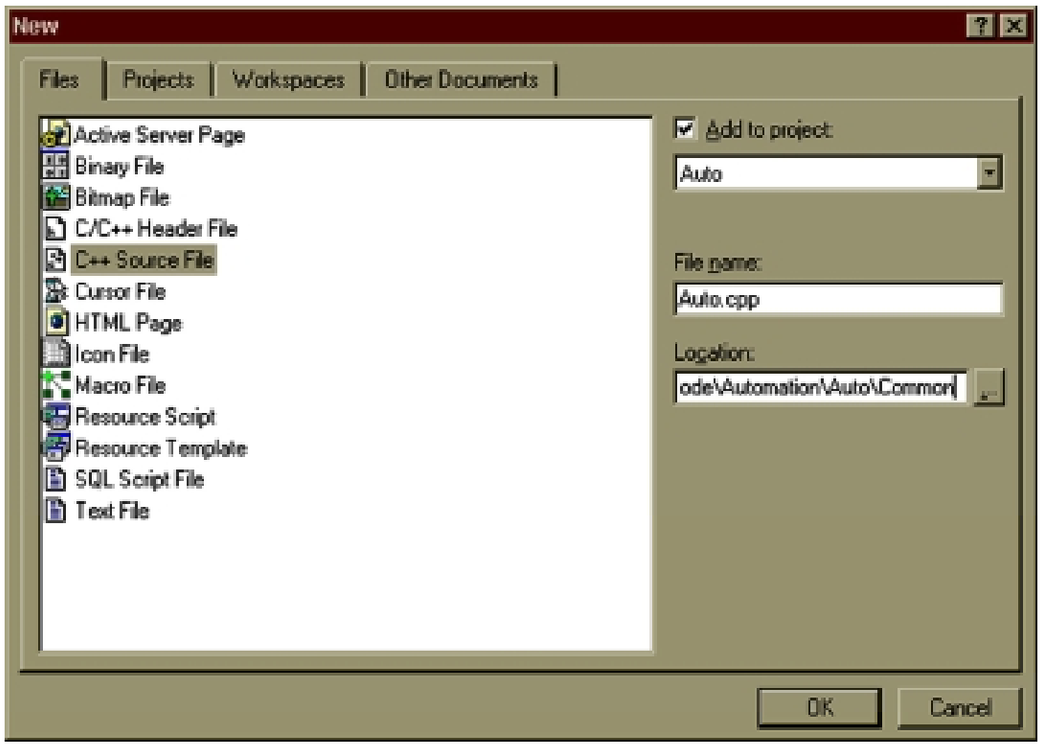


10. Click on "General" tab.
11. In the "Intermediate files:" field, type "temp".
12. Click "OK".

Adding The Main Source File

1. Choose from the menu, "File" > "New".

2. Select "Files" tab and choose "C++ Source File".
3. Ensure that the "Add to project:" is checked.
4. In the "File name:" field, type "Auto.cpp" into .
5. In the "Location:" field, type in the path similar to the following:
6. "D:\Adobe Photoshop 6.0 SDK\SampleCode\Automation\Auto\Common".
7. Click "OK".



8. In the "Auto.cpp" file, paste the contents of "Auto.cpp" located at the end of this section.
9. Change any occurrence of "Auto" to your plug-in name.
10. Save and close "Auto.cpp".

Adding Other Necessary Source Files

1. In the "FileView" window, right click on the "Source Files" folder.
2. Choose "Add Files to Folder..."
3. Browse to the following location in the SDK:
4. "D:\Adobe Photoshop 6.0 SDK\SampleCode\Common\Sources"
5. Double click "PIUGet.cpp".
6. Repeat the previous steps to add "PIUNew.cpp".
7. Repeat the previous steps to add "PIUSuites.cpp".

Building The Project

Your project settings and files are complete. You must verify that Auto can build.

1. Choose from the menu "Build" > "Build Auto.8li"

NOTE: If you are building the Release version, you need to choose from the menu "Build" > "Rebuild All".

2. Ensure that the output window ends with the following line: "Auto.8li - 0 error(s), 0 warning(s)." If not you probably have a typo somewhere. Go back through the steps and try again.

Creating A Plug-In Shortcut To Plug-Ins Folder

You need to get this binary into the Photoshop Plug-ins folder. That way, Photoshop loads the plug-in during the next execution of Photoshop. You can create a shortcut so that you can rebuild the plug-in and not have to copy and paste it into the Photoshop Plug-Ins folder each time.

1. The "Auto.8li" plug-in should be located in:

"D:\Adobe Photoshop 6.0 SDK\SampleCode\Output\Win\Debug\Auto.8li".

2. Right click on "Auto.8li", select "Copy"
3. Go to the "Plug-ins" folder for Photoshop, which may have a path similar to:
4. "D:\Photoshop\Plug-Ins"
5. Right click INSIDE the "Plug-Ins" folder, select "Paste Shortcut".

Loading The Plug-In Into Photoshop

1. Launch Photoshop.
2. Create a new document.
3. To verify that the "Auto" plug-in has an About box, choose from the menu "Help" > "About Plug-in" > "Auto".
4. To verify that Photoshop loaded the "Auto" plug-in properly, choose from the menu "File" > "Automate" > "Auto".

Congratulations!! Now, you have a functional Photoshop automation plug-in!

```
/****** Auto.r ********/

#include "PIDefines.h"
#ifdef __PIMac__
    #include "PIGeneral.r"
#elif defined(__PIWin__)
    #include "PIGeneral.h"
#endif

resource 'PiPL' ( 16000, "Auto", purgeable)
    {
        {
        Kind { Actions },//"Actions" for automation plug-ins
        Name { "Auto" },//name of your plug-in
        Category { "SDK Example" },//type of plug-in, ie. company
name
        Version { (latestActionsPlugInVersion << 16) | latestActions-
PlugInSubVersion },
        #ifdef __PIMac__
            CodePowerPC { 0, 0, "" },
        #elif defined(__PIWin__)
            CodeWin32X86 { "PluginMain" },//the name of the entrypoint
        #endif
    //  EnableInfo { "true" },//if true, enables plug-in when there's
//open document
                            //if unspecified, plug-in always enabled
        }
    };

/********end Auto.r********/

/****** Auto.cpp **********/

//-------------------------------------------------------------------
------------
//  Includes -- Use precompiled headers if compiling with CodeWar-
rior.
//-------------------------------------------------------------------
------------
#include <stdio.h>//defines sprintf
#include "PIUGet.h"//defines PIUGetInfo, not used in Template
#include "PIDefines.h"//defines __PIMac__, __PIWin__
#include "PIGeneral.h"//defines kPSPhotoshopCaller, kPSDoIt
#include "SPInterf.h"//defines kSPInterfaceCaller,
           //kSPInterfaceAboutSelector, kSPInterfaceStartupSelector,
```

```
#include "SPBasic.h"//defines SPBasicSuite
#include "SPAccess.h"//defines kSPAccessCaller,
       //kSPAccessReloadSelector, kSPAccessUnloadSelector,
       //not used in Template
#include "PIActions.h"//defines PSActionDescriptorProcs,
                  //PSActionReferenceProcs,
                         //PSActionControlProcs
#include "ADMBasic.h"//contains cross-platform alert dialogs,
                  //About boxes, file and directory dialogs,
                  //string conversions
#include "PIUSuites.h"//defines PIUSuitesAcquire and PIUSuitesRelease

#define DLLExport extern "C" __declspec(dllexport)


//-------------------------------------------------------------------
------------
//  Function Prototypes.
//-------------------------------------------------------------------
------------

DLLExport SPAPI SPErr PluginMain(const char* caller,
                                 const char* selector,
                                 const void* data );
//-------------------------------------------------------------------
------------
//  Globals -- Define global variables for plug-in scope.
//-------------------------------------------------------------------
------------

SPBasicSuite*sSPBasic = NULL;// the Basic Suite

//PSActionDescriptorProcs, PSActionReferenceProcs,
//sPSActionControlProcs
//are used in playing Photoshop events.
//Refer to MakeSelectFill sample plug-in for how the suites are used.
//create pointers to the suites
PSActionDescriptorProcs*sPSActionDescriptor = NULL;
PSActionReferenceProcs*sPSActionReference = NULL;
PSActionControlProcs*sPSActionControl = NULL;
PSActionListProcs*sPSActionList = NULL;
ADMBasicSuite*sADMBasic = NULL;

_AcquireList MySuites[] =
{
    kPSActionDescriptorSuite, kPSActionDescriptorSuiteVersion, (void
**)&sPSActionDescriptor,
    kPSActionReferenceSuite, kPSActionReferenceSuiteVersion, (void
**)&sPSActionReference,
    //note that the previous version of the Control suite
//is used for compatibility with 5.5
    kPSActionControlSuite, kPSActionControlSuitePrevVersion, (void
**)&sPSActionControl,
    kPSActionListSuite, kPSActionListSuiteVersion, (void **)&sPSAc-
tionList,
    kADMBasicSuite, kADMBasicSuiteVersion, (void **)&sADMBasic
};


//-------------------------------------------------------------------
------------
//
```

```
//  PluginMain
//
//  All calls to the plug-in module come through this routine.
//  It must be placed first in the resource. To achieve this,
//  most development systems require this be the first routine
//  in the source.
//
//  The entrypoint will be "pascal void" for Macintosh,
//  "void" for Windows.
//
//------------------------------------------------------------------------
------------

DLLExport SPAPI SPErr PluginMain(
                          const char* caller, // who is calling
                          const char* selector, // what do they want
                          const void* data )// what is the message
{
    SPErr error = kSPNoError;
    SPMessageData *basicMessage = NULL;

    //all messages contain a SPMessageData*
    basicMessage = (SPMessageData *) data;
    sSPBasic = basicMessage->basic;

    // acquire all the global suite pointers now
    error = PIUSuitesAcquire(sSPBasic,
                          MySuites,
                          sizeof (MySuites) / sizeof (_AcquireList));
    if (error) return error;

    // check for SP interface callers
    if (sSPBasic->IsEqual((char*)caller, kSPInterfaceCaller))
    {
        if (sSPBasic->IsEqual(
                    (char*)selector, kSPInterfaceAboutSelector))
        {
            //Pop the About box using sADMBasic
            sADMBasic->AboutBox(basicMessage->self,
                                "About Auto",
                                "Auto is an automation plug-in");
        }
    }
    // Photoshop is calling
    if (sSPBasic->IsEqual((char*)caller, kPSPhotoshopCaller))
    {
        // the one and only message
        if (sSPBasic->IsEqual((char*)selector, kPSDoIt))
        {
            //pop a dialog to show plug-in works
            sADMBasic->MessageAlert("Auto Automation Plug-In");
        }
    }
    // clean up, free memory
    PIUSuitesRelease(sSPBasic,
                     MySuites,
                     sizeof (MySuites) / sizeof (_AcquireList));
    return error;
}/*end PluginMain*/

// all windows plug-ins need a DllMain
```

```
#ifdef __PIWin__
BOOL APIENTRY DllMain( HANDLE hModule,
                           DWORD ul_rearson_for_call,
                           LPVOID lpReserved )
{
    return TRUE;
}
#endif

/********end Auto.cpp********/
```

Understanding PluginMain Function

The Photoshop plug-in manager communicates with your plug-in by loading the plug-in code into memory if necessary, then calling its entry point. By convention, the entry point is called "PluginMain":

```
DllExport SPAPI SPErr PluginMain( char *caller, char *selector, void
*data )
```

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. The caller string identifies the sender of the message and a general category of action. The selector string specifies the action to take within the category of action. All plug-ins receive five message actions: startup, shutdown, reload, unload and about. In addition, your plug-in may receive additional message actions specific to the plug-in type. The third parameter is a pointer to a data structure, which varies depending on the message action. Each time your plug-in is called, it receives a message action from Photoshop. The message action notifies your plug-in that an action has happened or tells your plug-in to perform an action.

Callers and Selectors

This section describes what the various caller/selector pairs mean and what your plug-in is expected to do in response to receiving them. Your plug-in's organization is largely based on the messages it receives. The main routine of your plug-in must first determine the message action using the caller and selector parameters.

Caller: kSPInterfaceCaller, Selector: kSPInterfaceStartupSelector

The first call a plug-in receives is kSPInterfaceStartupSelector. This is where the plug-in needs to allocate memory and add plug-in types to Adobe Photoshop.

Caller: kSPInterfaceCaller, Selector: kSPInterfaceShutdownSelector

The opposite of the startup selector is kSPInterfaceShutdownSelector. This selector is also received only once, when the Illustrator application is in the process of quitting. Actions that should happen when the user is completely finished using the plug-in, such as saving preference information, are done here. Also, any needed follow up action for something done during the kSPInterfaceStartupSelector message should be done here. Some actions, such as
adding a plug-in type, do not need any clean up.

Caller: kSPAccessCaller, Selector: kSPAccessReloadSelector

When the user triggers your plug-in, the plug-in receives the kSPAccess-Caller/kSPAccessReloadSelector message pair. Reload is your plug-inís opportunity to restore state information that it needs to run, such as global variables. Plug-in suites use the reload message to setup their function tables. Acquire suite calls should be made elsewhere.

Caller: kPSPhotoshopCaller, Selector: kPSDoIt

After the reload selector has been received, the plug-in receives the kPSPhotoshopCaller/kPSDoIt message pair. The core functionality of the plug-in should happen here.

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. As with the reload selector, you shouldnít use the unload selector as an opportunity to release suites.

Caller: kSPInterfaceCaller, Selector: kSPInterfaceAboutSelector

The kSPInterfaceCaller/kSPInterfaceAboutSelector message pair is received when the user selects from the menu "About Plug-in" > "plug-in name". This message gives your plug-in the opportunity to display a copyright or provide some other information about it's capabilities. Template uses ADM to display a simple alert dialog box. The dialog should be modal, and disappear when the user dismisses it.

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. While the contents of the message data varies, by convention all message data structures begin with three common fields, which are grouped into the SPMessageData structure:

```
typedef struct {
SPPluginRef self;
void *globals;
SPBasicSuite *basic;
} SPMessageData;
```

The self field is a reference to the plug-in being called. It is only needed when registering plug-in suites, adapters, and other plug-in data to Photoshop. The globals pointer is for use by your plug-in. It is used to preserve any information between calls that it needs, and is usually a pointer to a block of memory allocated by your plug-in at startup. The basic field is a pointer to the Basic Suite, which allows your plug-in to acquire and use other suites. The PSActionsPlugInMessage is the message you get when the caller/selector pair is kPSPhotoshopCaller/kPSDoIt. The actionParameters are used for scripting your plug-in. The following code is the structure of the PSActions-PlugInMessage and an example of how to cast the data message to "actions-Message".

```
typedef struct PSActionsPlugInMessage
    {
    SPMessageDatad;
    PIActionParameters*actionParameters;/* For recording and play-
back */
    } PSActionsPlugInMessage;

    if (sSPBasic->IsEqual (caller, kPSPhotoshopCaller) == 0)
```

```
actionsMessage = (PSActionsPlugInMessage *) data;
```

Photoshop Suites

The Photoshop 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 events within Photoshop. The mechanism for this is through 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. Photoshop's suite architecture, known as the Plug-in Component Architecture (PiCA) is also found in the latest versions of other Adobe applications. Conceptually, PiCA is similar to other software component architectures such as Microsoft's COM.

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 that the Photoshop 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 only "knows about" the Basic Suite, which is introduced earlier in this chapter (as part of the SPMessageData structure). The Basic Suite is used to acquire and release other suites. PSActionDescriptorProcs, PSActionReferenceProcs, PSActionListProcs, and PSActionControlProcs are the main suites used by automation plug-ins. PSActionDescriptorProcs contains the functions to make and free descriptors and access or verify keys within the descriptor. PSActionReferenceProcs contains the functions to make and free references, and put keys into a reference. PSActionListProcs contains functions that gets a list out of a descriptor, counts the number of elements in a list, and gets items out of a list. PSActionControlProcs defines the crucial Play function which plays the desired event.

Conclusion

At this point, you should be able to build a project workspace specifically for an automation plug-in. From previous chapters, you should know how to create functions based on the Listener plug-in output. From this chapter, build a automation plug-in project workspace. To get a good look at a fully functional plug-in that incorporates the concepts learned up to this point, go to the MakeSelectFill sample plug-in. MakeSelectFill shows the use and complete implementation of the MakeDoc, RectangularSelection, and ColorFill functions.

Perhaps, all the concepts covered from the beginning of the tutorial is not enough. Some automation goals depend on conditions of the Photoshop environment, such as the number of documents open or the dimensions of the document. Photoshop events can provide a variety of information about the application, document, layer, and more. You need to refer to the Get Info chapter which discusses getting information from Photoshop.

5 Get Info Functions

Introduction

Automation not only has the ability to play Photoshop events, but can get information from Photoshop, such as the number of documents open or the number of layers in a document. To get information from Photoshop, you need to specify what object you are talking about, i.e. a document or a layer. And what information about that object you are talking about, i.e. the number of layers or the layer name.
This chapter describes three example functions of getting information from Photoshop. The first example gets the number of documents currently open. The second example gets the pixel units of the document width. The third example gets the keywords from File Info of the document.

GetNumDocs Function

```
/*The GetNumDocs function gets the number of documents open.*/
SPErr GetNumDocs(int32* numDocs)
{
    PIActionReferencereference = NULL;
    PIActionDescriptorresult = NULL;
    Boolean        hasKey;
    SPErr error = kSPNoError;

    if (numDocs == NULL)
       return kSPBadParameterError;

    *numDocs = 0;
    error = sPSActionReference->Make( &reference );
    if (error) goto returnError;

    error = sPSActionReference->PutProperty(reference,
                                            classProperty,
                                            keyNumberOfDocuments );
    if (error) goto returnError;

    error = sPSActionReference->PutEnumerated(reference,
                                            classApplication,
                                            typeOrdinal,
                                            enumTarget);
    if (error) goto returnError;

    error = sPSActionControl->Get( &result,
                                   reference );
   if (error) goto returnError;

    error = sPSActionDescriptor->HasKey( result,
                                         keyNumberOfDocuments,
```

```
                                                &hasKey );
        if (hasKey == false) goto returnError;

        error = sPSActionDescriptor->GetInteger( result,
                                                 keyNumberOfDocuments,
                                                 numDocs );
        if (error) goto returnError;

        returnError:
        //free all descriptors and references
        if (reference != NULL) sPSActionReference->Free(reference);
        if (result != NULL) sPSActionDescriptor->Free(result);

        return error;

    }/*end GetNumDocs*/
```

Most of the information you need follows the same structure as the GetNumDocs function. The keyname that represents the number of documents is keyNumberOfDocuments. The object that contains keyNumberOfDocuments is the application. The corresponding object type is classApplication. The value type of keyNumberOfDocuments is integer. Therefore, the GetInteger function is used to get the value of keyNumberOfDocuments.

To create a function that gets information, you can copy the GetNumDocs function. Simply replace the classApplication, keyNumberOfDocuments, and GetInteger with the appropriate object type, keyname, and 'Get' function, respectively. You must refer to the tables in the following sections.

PIUGetInfo Function

If you do not want to create your own function, you may use PIUGetInfo function to get information. Given a class and an optional property PIUGetInfo returns the data you requested in "returnData". The optional "returnExtraData" holds the unit or string length information.

```
    SPErr PIUGetInfo(DescriptorClassID desiredClass,
                     DescriptorKeyID desiredKey,
                     void* returnData,
                     void* returnExtraData)
    {
        SPErr error = kSPNoError;
        PIActionReference reference = NULL;
        PIActionDescriptor result = NULL;
        PIActionDescriptor* data;

        if (returnData == NULL)
            error = kSPBadParameterError;
        if (error) goto returnError;

        error = sPSActionReference->Make(&reference);
        if (error) goto returnError;

        if (desiredKey)
        {
            error = sPSActionReference->PutProperty(reference,
                                                    classProperty,
                                                    desiredKey);
            if (error) goto returnError;
        }
```

```
        error = sPSActionReference->PutEnumerated(reference,
                                                  desiredClass,
                                                  typeOrdinal,
                                                  enumTarget);
        if (error) goto returnError;

        error = sPSActionControl->Get(&result,
                                      reference);
        if (error) goto returnError;

        if (desiredKey)
        {
            error = PIUGetSingleItemFromDescriptor(result,
                                                   desiredKey,
                                                   returnData,
                                                   returnExtraData);
            if (error) goto returnError;
        } else {
            data = (PIActionDescriptor*)returnData;
            *data = result;
            result = NULL; // you are responsible to free this guy
        }

       returnError:

        if (reference) sPSActionReference->Free(reference);
        if (result)sPSActionDescriptor->Free(result);

        return error;
    }
```

With the PIUGetInfo function, it is very easy to get the number of documents, as in the first example. You must simply make sure PIUGet.cpp is in your project and #include PIUGet.h in your source file. The following code is sufficient to get the number of documents open:

```
    /*Gets number of documents open*/
    SPErr GetnumDocs(void)
    {
       SPErr error = kSPNoError;
        int numDocs;
        error = PIUGetInfo(classApplication,
                           keyNumberOfDocuments,
                           &numDocs,
                           NULL);

        return error;
    }/*end GetnumDocs*/
```

GetDocWidth Function

The second example gets the pixel units of the document width. In the previous example, the fourth parameter is not used. The fourth parameter specifies the object type or units of the key value. For example, the units for "keyWidth" is unitPixels. You must pass unitPixels to the fourth parameter as illustrated in the following code.

```
    /*Gets pixel width of the current document*/
    SPErr GetDocWidth(void)
    {
       SPErr error = kSPNoError;
       double width;
```

```
        DescriptorUnitID unit = unitPixels;
        error = PIUGetInfo(classDocument,
                           keyWidth,
                           &width,
                           &unit);

        return error;
    }/*end GetDocWidth*/
```

Similarly, you can get the document height by replacing "keyWidth" with "keyHeight". Note: Photoshop only gives length in pixel units. You must do your own conversions by using the resolution.

GetKeywords Function

The third example for getting information is more complicated. The GetKeywords function gets the keyword list from the File Info dialog under the File menu ("File" > "File Info" > "Keywords"). The function displays each keyword in a dialog. In order to get the keywords, you need to get the FileInfo object, get the keyword list out of the FileInfo object, and finally get each keyword from the keyword list. Because keyKeywords contains a list of values, you can not model the function off of the GetNumDocs function.

```
    /*Gets keywords from the File Info and displays each keyword in a
    dialog*/
    SPErr GetKeywords(void)
    {
        SPErr error = kSPNoError;
        PIActionDescriptor fileInfo = NULL;
        PIActionList wordsList = NULL;
        DescriptorTypeID type = typeNull;//keyFileInfo is typeNull

        error = PIUGetInfo(classDocument,
                           keyFileInfo,
                           &fileInfo,
                           &type);
        if (error) goto returnError;
        error = sPSActionDescriptor->GetList(fileInfo,
                                             keyKeywords,
                                             &wordsList);
        if (error) goto returnError;
        if (wordsList != NULL && error == 0)
        {
            uint32 listCount = 0;
            sPSActionList->GetCount(wordsList,
                                    &listCount);
            //indexing of list starts at 0
            for (uint32 counter = 0; counter < listCount; counter++)
            {
                uint32 stringLen = 0;
                char* localString;

                error = sPSActionList->GetStringLength(wordsList,
                                                       counter,
                                                       &stringLen);
                if (error) goto returnError;
                if (error == 0 && stringLen != 0)
                {
                   // add one for '\0'
                    stringLen++;
```

```
                // Make some room for the data, then dispose it later
                localString = new char[stringLen];
                if (localString != NULL)
                {
                    error = sPSActionList->GetString(wordsList,
                                                     counter,
                                                     localString,
                                                     stringLen);
                    if (error) goto returnError;
                    //pop dialog that shows keyword
                    else sADMBasic->MessageAlert(localString);
                    delete [] localString;
                }
            }
        }
    }/*end if (wordsList != NULL && error == 0)*/

    returnError:

    //free all descriptors and references
    if (fileInfo != NULL) sPSActionDescriptor->Free(fileInfo);
    if (wordsList != NULL) sPSActionList->Free(wordsList);
    return error;

}/*end GetKeywords*/
```

Conceptually, the third example is the same as the first. The PIUGetInfo function is used to get FileInfo object. The sPSActionDescriptor->GetList function gets the list of keywords out of FileInfo. The GetString function gets each keyword, "localString" out of "wordsList". Note: Refer to PIActions.h for the function prototypes for GetFloat, GetInteger, GetString, GetObject, and other 'Get' functions.

Common GetInfo Functions

IsDirty Function

```
/*The IsDirty function checks if doc was modified since last save*/
SPErr IsDirty(uint8* pDirty)
{
    PIActionReferencereference = NULL;
    PIActionDescriptorresult = NULL;
    Boolean        hasKey;
    SPErr error = kSPNoError;

    if (pDirty == NULL)
       return kSPBadParameterError;

    *pDirty = 0;//set document to not dirty
    error = sPSActionReference->Make( &reference );
    if (error) goto returnError;

    error = sPSActionReference->PutProperty(reference,
                                            classProperty,
                                            keyIsDirty );
    if (error) goto returnError;

    error = sPSActionReference->PutEnumerated(reference,
                                              classDocument,
```

```
                                                typeOrdinal,
                                                enumTarget);
    if (error) goto returnError;

    error = sPSActionControl->Get( &result,
                                   reference );
   if (error) goto returnError;

    error = sPSActionDescriptor->HasKey( result, keyIsDirty, &hasKey
);
    if (hasKey == false) goto returnError;

    error = sPSActionDescriptor->GetBoolean( result,
                                                keyIsDirty,
                                                pDirty );
    if (error) goto returnError;

    returnError:
    //free all descriptors and references
    if (reference != NULL) sPSActionReference->Free(reference);
    if (result != NULL) sPSActionDescriptor->Free(result);

    return error;

}/*end IsDirty*/
```

GetLayerIndex Function

```
//Gets the index of the current layer.  Index starts at 1
SPErrGetLayerIndex( int32* layerIndex )
{
   SPErr error = kSPNoError;

    PIActionReferenceref = NULL;
    PIActionDescriptorresult = NULL;
    Boolean        hasKey;

    error = sPSActionReference->Make( &ref );
    if (error) goto returnError;

    error = sPSActionReference->PutProperty( ref,
                                                classProperty,
                                                keyItemIndex );
    if (error) goto returnError;

    error = sPSActionReference->PutEnumerated(ref,
                                                classLayer,
                                                typeOrdinal,
                                                enumTarget );
    if (error) goto returnError;

    error = sPSActionControl->Get( &result,
                                      ref );
    if (error) goto returnError;

   error = sPSActionDescriptor->HasKey( result,
                                           keyItemIndex,
                                           &hasKey );
    if ( error || hasKey == false ) goto returnError;

    error = sPSActionDescriptor->GetInteger( result,
```

```
                                                keyItemIndex,
                                                layerIndex );
    if (error) goto returnError;

     returnError:
     if ( ref ) sPSActionReference->Free ( ref );
     if ( result ) sPSActionDescriptor->Free ( result );
     return error;

}/*end GetLayerIndex*/
```

GetDocIndex Function

```
/*Get index of current document and pops dialog showing document
index.  Indexing starts at 1*/
SPErr GetDocIndex (int32* docIndex)
{
    SPErr error = kSPNoError;
    PIActionReferenceref = NULL;
    PIActionDescriptorresult = NULL;

    error = sPSActionReference->Make( &ref );
    if(error) goto returnError;

    error = sPSActionReference->PutEnumerated( ref,
                                               classDocument,
                                               typeOrdinal,
                                               enumTarget );
   if(error) goto returnError;
    error = sPSActionControl->Get( &result,
                                   ref );
   if(error) goto returnError;

    *docIndex = -1;
    error = sPSActionDescriptor->GetInteger(result,
                                            keyItemIndex,
                                            docIndex);

    //pop dialog that shows ID of currently selected document
    char string[81];
    char *text;
    text = string;
    error = sADMBasic->ValueToString((float)*docIndex,
                                     text,
                                     256,
                                     kADMNoUnits,
                                     2,
                                     false);
   if(error) goto returnError;
    error = sADMBasic->MessageAlert(string);
    if(error) goto returnError;

    returnError:
    /*free descriptors and references*/
    if (ref != NULL)sPSActionReference->Free(ref);
    if (result != NULL)sPSActionDescriptor->Free(result);
    return error;

}/*end GetDocIndex*/
```

CloseFile Function

```
/*Close current file*/
SPErr CloseFile(void)
{
    SPErr error = kSPNoError;
    PIActionDescriptor result;
    PIActionDescriptor desc;

   error = sPSActionDescriptor->Make(&desc);
   if(error) goto returnError;

    error = sPSActionDescriptor->PutEnumerated(desc,
                                               keySaving,
                                               typeYesNo,
                                               enumNo);
    if(error) goto returnError;

    error = sPSActionControl->Play(&result,
                                   eventClose,
                                   desc,
                                   plugInDialogSilent);
    if(error) goto returnError;

    returnError:
   /*free descriptors and references*/
    if (result != NULL)sPSActionDescriptor->Free(result);
   if (desc != NULL)sPSActionReference->Free(desc);
    return error;

}/*end CloseFile*/
```

CopyLayer Function

```
/*Copy current layer of current document*/
SPErr CopyLayer(void)
{
   SPErr error = kSPNoError;
   PIActionDescriptor desc = NULL;
   PIActionReference ref = NULL;
   PIActionDescriptor resultSet = NULL;
   PIActionDescriptor resultCopy = NULL;

    error = sPSActionDescriptor->Make(&desc);
   if (error) goto returnError;

    error = sPSActionReference->Make(&ref);
    if (error) goto returnError;

    /*select contents of currently selected layer*/
   error = sPSActionReference->PutProperty(ref,
                                           classChannel,
                                           keySelection);
   if (error) goto returnError;

   error = sPSActionDescriptor->PutReference(desc,
                                             keyNull,
                                             ref);
   if (error) goto returnError;

   error = sPSActionDescriptor->PutEnumerated(desc,
                                              keyTo,
```

```
                                                typeOrdinal,
                                                enumAll);
    if (error) goto returnError;

    error = sPSActionControl->Play(&resultSet,
                                   eventSet,
                                   desc,
                                   plugInDialogSilent);
    if (error) goto returnError;

    /*copy layer*/
     error = sPSActionControl->Play(&resultCopy,
                                    eventCopy,
                                    NULL,
                                    plugInDialogSilent);
     if (error) goto returnError;

     returnError:
    /*free descriptors and references*/
     if (desc != NULL)sPSActionDescriptor->Free(desc);
     if (ref != NULL)sPSActionReference->Free(ref);
     if (resultSet != NULL)sPSActionDescriptor->Free(resultSet);
     if (resultCopy != NULL)sPSActionDescriptor->Free(resultCopy);
     return error;

}/* end CopyLayer*/
```

OpenFile Function

```
/*Open file that is located at "filepath"*/
SPErr OpenFile(char* filepath)
{
   SPErr error = kSPNoError;
   Handle  tempAlias;
   PIActionDescriptor result = NULL;
    PIActionDescriptor desc = NULL;

     #if Macintosh
        NewAliasMinimalFromFullPath(strlen(filepath),
                                    filepath,
                                    nil,
                                    nil,
                                    &(AliasHandle)
 tempAlias);
     #else
        size_t size = strlen(filepath);
        tempAlias = sPSHandle->New(size+1);
        if (tempAlias != NULL)
        {
           GlobalLock(tempAlias);
            strncpy(*tempAlias,
                    filepath,
                    strlen(filepath)+1);
            GlobalUnlock(tempAlias);
        }
   #endif


    error = sPSActionDescriptor->Make(&desc);
    if (error) goto returnError;
    error = sPSActionDescriptor->PutAlias(desc,
```

```
                                        keyNull,
                                        tempAlias);
    if (error) goto returnError;
    error = sPSActionControl->Play(&result,
                                   eventOpen,
                                   desc,
                                   plugInDialogSilent);
    if (error) goto returnError;

    returnError:
    #if Macintosh
        if (tempAlias != NULL) DisposeHandle(tempAlias);
    #else
        if (tempAlias != NULL) sPSHandle->Dispose(tempAlias);
    #endif
    /*free descriptors and references*/
    if (desc != NULL)error = sPSActionDescriptor->Free(desc);
    if (result != NULL)error = sPSActionDescriptor->Free(result);
    return error;

}/* end OpenFile */
```

SaveFileAsEPS Function

```
/*Saves the given file as EPS.
typeEPSPreview can be enumTIFF or enumMacintosh
typeDepth can be enum1BitPerPixel or enum8BitsPerPixel
typeEncoding can be enumASCII, enumBinary, enumJPEG, or enumZip*/

SPErr SaveFileAsEPS (char* filePath, DescriptorEnumID preview,
DescriptorEnumID depth, DescriptorEnumID encoding,
boolean halftoneScreen, boolean transferFunction,
boolean colorManagement, boolean copy, boolean lowercase)
{
   SPErr error = kSPNoError;
   PIActionDescriptor result = NULL;
   PIActionDescriptor desc = NULL;
   PIActionDescriptor EPSDescriptor = NULL;
   Handle  tempAlias = NULL;

   #if Macintosh
        NewAliasMinimalFromFullPath(strlen(filePath),
                                   filePath,
                                   nil,
                                   nil,
                                   &(AliasHandle)tempAlias);
   #else
      tempAlias = sPSHandle->New(strlen(filePath)+1);
      if (tempAlias != NULL)
      {
           GlobalLock(tempAlias);
           strncpy(*tempAlias,
                   filePath,
                   strlen(filePath)+1);
           GlobalUnlock(tempAlias);
      }
   #endif

    error = sPSActionDescriptor->Make(&desc);
    if(error) goto returnError;
```

```
error = sPSActionDescriptor->Make(&EPSDescriptor);
if(error) goto returnError;

error = sPSActionDescriptor->PutEnumerated(EPSDescriptor,
                                           keyPreview,
                                           typeEPSPreview,
                                           preview);
if(error) goto returnError;
error = sPSActionDescriptor->PutEnumerated(EPSDescriptor,
                                           keyDepth,
                                           typeDepth,
                                           depth);
if(error) goto returnError;
error = sPSActionDescriptor->PutEnumerated(EPSDescriptor,
                                           keyEncoding,
                                           typeEncoding,
                                           encoding);
if(error) goto returnError;
error = sPSActionDescriptor->PutBoolean(EPSDescriptor,
                                        keyHalftoneScreen,
                                        halftoneScreen);
if(error) goto returnError;
error = sPSActionDescriptor->PutBoolean(EPSDescriptor,
                                        keyTransferFunction,
                                        transferFunction);
if(error) goto returnError;
error = sPSActionDescriptor->PutBoolean(EPSDescriptor,
                                        keyColorManagement,
                                        colorManagement);
if(error) goto returnError;
error = sPSActionDescriptor->PutObject(desc,
                                       keyAs,
                                       classPhotoshopEPSFormat,
                                       EPSDescriptor);
if(error) goto returnError;
error = sPSActionDescriptor->PutAlias(desc,
                                      keyIn,
                                      tempAlias);
if(error) goto returnError;
error = sPSActionDescriptor->PutBoolean(desc,
                                        keyCopy,
                                        copy);
if(error) goto returnError;
error = sPSActionDescriptor->PutBoolean(desc,
                                        keyLowerCase,
                                        lowercase);
if(error) goto returnError;
error = sPSActionControl->Play(&result,
                               eventSave,
                               desc,
                               plugInDialogSilent);
if(error) goto returnError;

returnError:
 /*free descriptors and references*/
 if (desc != NULL)
     error = sPSActionDescriptor->Free(desc);
 if (EPSDescriptor != NULL)
     error = sPSActionDescriptor->Free(EPSDescriptor);
 if (result != NULL)
     error = sPSActionDescriptor->Free(result);
```

```
    #if Macintosh
        if (tempAlias != NULL)  DisposeHandle( tempAlias );
    #else
        if (tempAlias != NULL)  sPSHandle->Dispose(tempAlias);
    #endif
    return error;

}/*end SaveFileAsEPS*/
```

PlayPluginDissolveSans Function

```
/*Calls a the DissolveSansAppleScript plug-in*/
SPErr PlayPluginDissolveSans(void)
{
   SPErr error = kSPNoError;
   PIActionDescriptor result = NULL;
   PIActionDescriptor descriptor = NULL;
   DescriptorTypeID runtimeID = 0;

   error = sPSActionDescriptor->Make(&descriptor);
   if (error) goto returnError;

   //put values necessary for the plugin into the descriptor, based
on #define keys
   error = sPSActionDescriptor->PutUnitFloat(descriptor,
                                             keyAmount,
                                             unitPercent,
                                             100);
   if (error) goto returnError;
   error = sPSActionDescriptor->PutEnumerated(descriptor,
                                              'disP',
                                              'mooD',
                                              'moD1');
   if (error) goto returnError;
    error = sPSActionDescriptor->PutUnitFloat(descriptor,
                                              keyAmount,
                                              unitPercent,
                                              100);
    if (error) goto returnError;

    //enter the name of the plugin as written in the PiPL
   //#define vendorName"AdobeSDK"
    //#define plugInName"Dissolve-sans-AppleScript"
   error = sPSActionControl->StringIDToTypeID(
                              "AdobeSDK Dissolve-sans-AppleScript",
                              &runtimeID);
    error = sPSActionControl->Play(&result,
                                   runtimeID,
                                   descriptor,
                                   plugInDialogSilent);

    returnError:
    if (result != NULL)
        error = sPSActionDescriptor->Free(result);
    if (descriptor != NULL)
        error = sPSActionDescriptor->Free(descriptor);
    return error;

}/*end PlayPluginDissolveSans*/
```

GetDocumentID Function

```
/*Gets document ID, a unique integer for each document per Photoshop
session.  Pops dialog showing document ID*/
SPErr GetDocumentID (int32* documentID)
{
    SPErr error = kSPNoError;
    PIActionReferenceref = NULL;
    PIActionDescriptorresult = NULL;


    error = sPSActionReference->Make( &ref );
    if(error) goto returnError;
    error = sPSActionReference->PutEnumerated( ref,
                                               classDocument,
                                               typeOrdinal,
                                               enumTarget );
    if(error) goto returnError;
    error = sPSActionControl->Get( &result,
                                   ref );
    if(error) goto returnError;

   error = sPSActionDescriptor->GetInteger(result,
                                           keyDocumentID,
                                           documentID);
   if(error) goto returnError;

    //pop dialog that shows ID of currently selected document
    char string[81];
    char *text;
    text = string;
    error = sADMBasic->ValueToString((float)*documentID,
                                     text,
                                     256,
                                     kADMNoUnits,
                                     2,
                                     false);
    if(error) goto returnError;
    error = sADMBasic->MessageAlert(string);
    if(error) goto returnError;

    returnError:
    if ( ref ) sPSActionReference->Free ( ref );
    if ( result ) sPSActionDescriptor->Free ( result );
    return error;

}/*end GetDocumentID*/
```

Get Info Classes, Types, and Keys

Below are tables of the valid classes for the PIUGetInfo routine. The Key column are valid values for the second parameter. The Type column is the type of variable you should pass in for the third parameter. The fourth parameter is required for Enumerated, UnitFloat, String, Object and GlobalObject requests, check the Comments field in the tables. Note that the string data requires a previously allocated array.

NOTE: For classApplication "colorSettings" you must call StringIDToTypeID to get the runtime key for "colorSettings".

Table 5–1: classApplication

Key	Type	Comments
keyRulerUnits	DescriptorEnumID	4th parameter required of type DescriptorEnum-TypeID
keyExactPoints	Boolean	
keyNumberOfCacheLevels	int32	
keyUseCacheForHistograms	Boolean	
keyInterpolationMethod	DescriptorEnumID	4th parameter required of type DescriptorEnum-TypeID
keyShowToolTips	Boolean	
keyPaintCursorKind	DescriptorEnumID	4th parameter required of type DescriptorEnum-TypeID
keySerialString	char[]	4th parameter required of type uint32, length of string returned
keyHostName	char []	4th parameter required of type uint32, length of string returned
keyGridMajor	int32	
keyGridMinor	int32	
keyWatchSuspension	int32	
keyNumberOfDocuments	int32	
keyInterfaceWhite	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceButtonUpFill	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceBevelShadow	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceIconFillActive	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceIconFillDimmed	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfacePaletteFill	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceIconFrameDimmed	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceIconFrameActive	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceBevelHighlight	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceButtonDownFill	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceIconFillSelected	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceBorder	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceButtonDarkShadow	PIActionDescriptor	4th parameter required of type DescriptorClassID

Key	Type	Comments
keyInterfaceIconFrameSelected	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceRed	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceBlack	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceToolTipBackground	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceToolTipText	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceTransparencyFore-ground	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyInterfaceTransparencyBack-ground	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyHistoryPrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyHostVersion	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyEyeDropperSample	DescriptorEnumID	4th parameter required of type DescriptorEnum-TypeID
keyColorPickerPrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyGeneralPrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyFileSavePrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyDisplayPrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyTransparencyPrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyUnitsPrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyGuidesPrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyPluginPrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyCachePrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyTransparencyGrid	DescriptorEnumID	4th parameter required of type DescriptorEnum-TypeID
keyCachePrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
"colorSettings" see NOTE above	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyForegroundColor	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyBackgroundColor	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyCachePrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID

Key	Type	Comments
keyCachePrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyCachePrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyCachePrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyCachePrefs	PIActionDescriptor	4th parameter required of type DescriptorClassID

Table 5–2: classDocument

Key	Type	Comments
keyMode	DescriptorEnumID	4th parameter required of type DescriptorEnum-TypeID
keyBigNudgeH	int32	
keyBigNudgeV	int32	
keyRulerOriginH	int32	
keyRulerOriginV	int32	
keyWidth	int32	4th parameter required of type DescriptorUnitID
keyHeight	int32	4th parameter required of type DescriptorUnitID
keyResolution	int32	4th parameter required of type DescriptorUnitID
keyTitle	char []	4th parameter required of type uint32, length of string returned
keyFileInfo	PIActionDescriptor	4th parameter required of type DescriptorClassID
keyNumberOfPaths	int32	
keyNumberOfChannels	int32	
keyNumberOfLayers	int32	
keyTargetpathIndex	int32	
keyWorkPathIndex	int32	
keyClippingPathInfo	PIActionDescriptor	44th parameter required of type DescriptorClassID
keyDepth	int32	
keyFileReference	Handle	
keyDocumentID	int32	
keyCopyright	Boolean	
keyWatermark	Boolean	
keyIsDirty	Boolean	
keyCount	int32	
keyItemIndex	int32	

Table 5–3: classLayer

Key	Type	Comments
keyName	char []	4th parameter required of type uint32, length of string returned
keyColor	DescriptorEnumID	4th parameter required of type DescriptorEnum-TypeID
keyVisible	Boolean	
keyMode	DescriptorEnumID	4th parameter required of type DescriptorEnum-TypeID
keyOpacity	int32	
keyLayerID	int32	
keyItemIndex	int32	
keyCount	int32	
keyPreserveTransparency	Boolean	
keyLayerFXVisible	Boolean	
keyGlobalAngle	int32	
keyBackground	Boolean	

Table 5–4: classBackgroundLayer

Key	Type	Comments
keyName	char []	4th parameter required of type uint32, length of string returned
keyColor	DescriptorEnumID	4th parameter required of type DescriptorEnum-TypeID
keyVisible	Boolean	
keyMode	DescriptorEnumID	4th parameter required of type DescriptorEnum-TypeID
keyOpacity	int32	
keyLayerID	int32	
keyItemIndex	int32	
keyCount	int32	
keyPreserveTransparency	Boolean	
keyLayerFXVisible	Boolean	
keyGlobalAngle	int32	
keyBackground	Boolean	

Table 5–5: classChannel

Key	Type	Comments
keyChannelName	char []	4th parameter required of type uint32, length of string returned
keyItemIndex	int32	
keyCount	int32	
keyVisible	Boolean	

Table 5–6: classAction

Key	Type	Comments
keyName	char []	4th parameter required of type uint32, length of string returned
keyItemIndex	int32	
keyCount	int32	
keyNumberOfChildren	int32	
keyParentName	char []	4th parameter required of type uint32, length of string returned
keyParentIndex	int32	

Table 5–7: classActionSet

Key	Type	Comments
keyName	char []	4th parameter required of type uint32, length of string returned
keyItemIndex	int32	
keyCount	int32	
keyNumberOfChildren	int32	

Table 5–8: classWorkPath

Key	Type	Comments
keyPathName	char []	4th parameter required of type uint32, length of string returned
keyPathContents	PIActionDescriptor	44th parameter required of type DescriptorClassID
keyItemIndex	int32	
keyCount	int32	
keyKind	DescriptorEnumID	4th parameter required of type DescriptorEnum-TypeID
keyTargetPath	Boolean	

Table 5–9: classPath

Key	Type	Comments
keyPathName	char []	4th parameter required of type uint32, length of string returned
keyPathContents	PIActionDescriptor	44th parameter required of type DescriptorClassID
keyItemIndex	int32	
keyCount	int32	
keyKind	DescriptorEnumID	4th parameter required of type DescriptorEnum-TypeID
keyTargetPath	Boolean	

Table 5–10: classHistory

Key	Type	Comments
keyName	char []	4th parameter required of type uint32, length of string returned
keyItemIndex	int32	
keyCount	int32	
keyAuto	Boolean	
keyHistoryBrushSource	Boolean	
keyCurrentHistoryState	Boolean	

Conclusion

You are now ready to incorporate the "Get" functions into a project workspace. The easiest approach is to paste your function definitions in the source file of your project then make function calls from the PluginMain function. You may use the Template sample plug-in as a starting point. You may use the MakeSelectFill sample plug-in as a reference to how functions are incorporated into an automation plug-in project.