ADOBE® CREATIVE SUITE® 3

# ADOBE DIALOG MANAGER PROGRAMMER'S GUIDE

Adobe®

*Adobe Dialog Manager Programmer's Guide*

# Contents

# Preface

Adobe® Dialog Manager (ADM) is a collection of platform-independent APIs for displaying and controlling dialog boxes.

## What is in this guide

This document provides an overview of the window and control architecture, describes how to use Adobe Dialog Manager with several Adobe products, and has individual chapters for each API suite. Each suite chapter contains an introduction to the suite, discusses concepts and structures used by the suite, and provides references for details about suite functions.

## Intended audience

This guide is intended for experienced developers who are familiar with the architecture of the product for which dialog boxes are to be added or modified.

## Typographic conventions

The following typographic conventions are used in this document:

| | |
|---|---|
| `Monospaced font` | Literal values and code, like JavaScript code, HTML code, filenames, and path names. |
| *`Italicized monospaced font`* | Variables or placeholders in code. For example, in `name="`*`myName`*`"`, the text *`myName`* represents a value you are expected to supply, such as `name="Fred"`. This also highlights the first occurrence of a new term. |
| *Italics* | Items to be emphasized, such as the first occurrence of a new term. |
| Blue underlined text | A hyperlink you can click to go to a related section in this book or a URL in your Web browser. |
| > | A shorthand notation for navigating to menu items; for example, Edit > Cut refers to the Cut item in the Edit menu. |

## Related documentation

For detailed information about the ADM APIs, see *Adobe Illustrator CS3 SDK API Reference*.

# 1 ADM Overview

Adobe® Dialog Manager (ADM) is a cross-platform API for implementing dialog interfaces for Adobe applications such as Adobe Photoshop® and Adobe Illustrator®. This document describes ADM structures and how to access them. Before reading it, you should already be familiar with the concept of dialogs and dialog items.

ADM enables developers to create and manage cross-platform dialogs. Two types of dialogs are supported:

- *Modal* dialogs are displayed on user input and disappear on the conclusion of the user input. With a modal dialog, a user cannot work elsewhere in the application until the dialog is closed.
- *Modeless* dialogs "float" over the host application windows.

In both cases, ADM supports many control types, including basic ones like buttons and text and more complicated ones like lists and hierarchy lists. In addition to this wide array of custom and standard user-interface elements, ADM provides behaviors like tab panels, docking panels, and automatically tracking and displaying the correct selection in grouped radio buttons. Finally, ADM provides a consistent Adobe interface and "look and feel."

ADM is implemented as a PICA plug-in and uses the PICA suites to export its functionality. A *plug-in* is any file containing a computer program and resources that extend the function of the host application. For more information on PICA, see "PICA plug-ins" on page 9.

Basic ADM functionality is provided using three core function suites:

- **ADM Basic suite** — Basic user interactions and utilities.
- **ADM Dialog suite** — Creating and managing dialogs.
- **ADM Item suite** — Creating and managing items in a dialog.

Several other suites provide other behaviors and allow ADM's functionality to be extended to cover many different custom interfaces. C or C++ interfaces can be used for each suite.

## Conventions

In this guide, constants are denoted with a preceding lowercase k (e.g., kADMClippedTextStaticStyle). The capital letters ADM in a suite name mean the suite is provided by ADM. For more information on terms, see "Glossary."

By convention, pointers to suites are named as follows:

```
ADMBasicSuite *sADMBasic;
ADMDialogSuite *sADMDialog;
ADMDialogGroupSuite *sADMDialogGroup;
ADMDrawerSuite *sADMDrawer;
ADMEntrySuite *sADMEntry;
ADMHierarchyListSuite *sADMHierarchyList;
ADMIconSuite *sADMIcon;
ADMImageSuite *sADMImage;
ADMItemSuite *sADMItem;
ADMListSuite *sADMList;
ADMListEntrySuite *sADMListEntry;
ADMNotifierSuite *sADMNotifier;
ADMTrackerSuite *sADMTracker;
```

# Accessing suites

Suites can be accessed through the use of predefined constants for the suite and suite version and the previously defined suite pointer. For example:

```
ADMDialogSuite *sADMDialog;
error = sSPBasic->AcquireSuite(kADMDialogSuite, kADMDialogSuiteVersion2, &sADMDialog);
if (error) goto . . . //handle error
```

In the code above, the `sSPBasic` variable is assigned a pointer when your plug-in loads. This pointer enables access to a data structure that enables access to the suites. Some applications may provide other, more transparent methods for obtaining suites through their own APIs.

# Architecture

## PICA plug-ins

PICA is an Adobe standard plug-in architecture used by several Adobe applications, including Photoshop and Illustrator. PICA provides a common plug-in management core to the host application and a standard interface for plug-ins. In Adobe documentation and header files, PICA often is referred to as Sweet Pea, SuitePea, SweetPEA, SuiteP, etc.; all these terms are synonymous with PICA.

The ADM application programming interface (API) is exposed to the host and its plug-in's via "suites." A suite is simply a pointer to a data structure that provides an interface to a common object, often a collection of function pointers (e.g., a group of functions to access an ADM Dialog object). Plug-ins can extend the host API by providing their own function suites.

Before they can be used, all suites must be "acquired"; when no longer needed, suites are "released." This mechanism guarantees the functions always are available to the plug-in.

An acquired suite actually is a pointer to a structure with the suite's function pointers. To call one of the suite functions, the syntax is as follows:

```
sSuite->function();
```

So, to use a suite function, you do something like this:

```
SPBasicSuite *sSPBasic = message->basic;
ADMBasicSuite *sADMBasic;

sSPBasic->AcquireSuite(kADMBasicSuite,  kADMBasicSuiteVersion2, &sADMBasic);
sADMBasic->Beep();
sBasic->ReleaseSuite(kADMBasicSuite,  kADMBasicSuiteVersion2);
```

The convention used by most SDKs is for suite variables to be global in scope and indicated by a lowercase "s" followed by the suite name; e.g., `sADMBasic`, as shown above.

Typically, the version number parameter you pass to `AcquireSuite` should be the version that contains the functions you need. All available suite versions are contained in the corresponding ADM header file (for example, `ADMBasic.h`), so you can include this header in any project you are writing.

**NOTE:** Do not assume higher-numbered versions of the product are supersets of lower numbered versions—they may not be.

PICA plug-ins are called by the application at certain times. A PICA event is received through the plug-in's main entry point, which is defined as follows:

```
ASAPI ASErr PluginMain(char *caller, char *selector, void *message);
```

The caller and selector indicate the type of event. The message is a pointer to a structure with any data necessary to handle the event. The ADM message structure always contains the following data:

```
typedef struct SPMessageData {

    long SPCheck;/* kSPValidSPMessageData if a valid SPMessage */
    struct SPPlugin *self;  /* SPPluginRef */
    void *globals;
    struct SPBasicSuite *basic;

} SPMessageData;
```

Plug-ins also might be called through callbacks they give to some host, like the application or ADM. In this case, it is the caller's responsibility to specify what information is available and provide enough information for the plug-in to work.

PICA plug-ins are loaded into and unloaded from memory as needed. When a PICA plug-in adds an ADM dialog, it remains in memory until the dialog is disposed of (for PICA version 2.4 and later; earlier versions of PICA require the plug-in to acquire itself, to remain in memory).

Some SDKs, like the Acrobat SDK, provide special code that handles suite acquisition and release automatically, so the programmer does not need to worry about these details.

## ADM objects

ADM user interfaces are built out of ADM user-interface objects. These objects include the dialog windows (dialog objects) and the dialog items (item objects) within the windows, as shown in the following figure:

A plug-in or application using ADM has access to standard ADM dialog types (modal and non-modal) and items (buttons and other user-interface controls). The user interface can be built in code or by using resource definitions. For instance, standard platform resources can be used to define the layout of user-interface objects. All objects have *properties* and *events* that determine their default behavior and allow them to be modified or extended. These also can be set in code or via a resource.

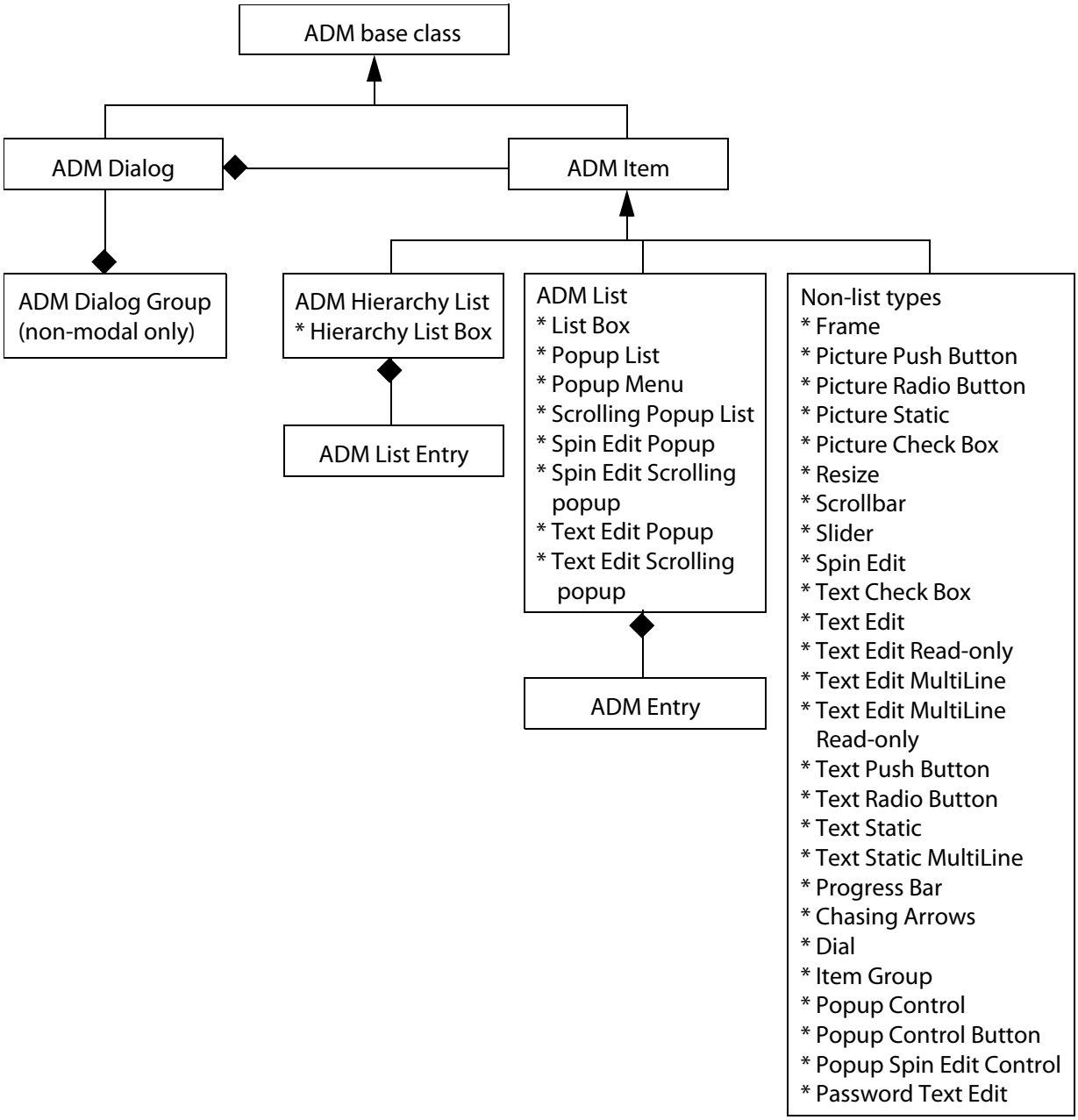
ADM has an object-oriented design, even though its interfaces are exported as procedural C functions. This is important, since many of the properties, behaviors, and callback functions of the various types of ADM user-interface objects (dialogs or dialog items) are the same. Understanding the fundamentals of managing one type of ADM user-interface object results in understanding how to manipulate other ADM objects as well.

For instance, ADM objects have associated text. For ADM windows, this is the window title; for a button, the button title; for a text-edit item, the editable text entered by the user. To access any ADM user-interface item's text, use these two functions:

```
void ADMAPI (*SetText)(ADMItemRef inItem, const char* inText);
void ADMAPI (*GetText)(ADMItemRef inItem, char* outText, ADMInt32 inMaxLen);
```

Some ADM objects need additional support functions or properties. For instance, a window object has functions to perform operations like setting the minimum and maximum window size. ADM text-edit items have additional functions to support properties like justification and numeric precision.

The complete ADM object hierarchy is shown in the following figure.




ADM List objects are owned by List Box, Popup List, Popup Menu, Scrolling Pop List, Spin Edit Popup, Spin Edit Scrolling Popup, Text Edit Popup, and Text Edit Scrolling Popup items. ADM Entry objects are contained by an ADM List.

## Quick summary of using ADM

To use ADM, first use a platform-specific resource editor to add a dialog resource to your plug-in project. At an appropriate point in your plug-in code (likely responding to an application API event), create a new ADM dialog with either `sADMDialog->Modal` (to create a modal dialog) or `sADMDialog->Create` (to create a non-modal, or floating or tabbed, dialog). Pass this function an initialization function that is called after ADM loads the resources and creates the dialog. Use this opportunity to set initial values or otherwise customize the dialog's behavior. ADM provides several suites of functions for accessing ADM objects, and these are used to perform the initialization.

ADM then displays and handles the dialog for you, processing user events as needed. Your application is called to handle certain standard events, like closing the dialog, and any other events requested in your init function, like notification that a button was pressed. For non-modal dialogs, call the ADM Dialog suite function `sADMDialog->Destroy` when the dialog is no longer needed. ADM destroys the dialog and frees its resources.

## Types

The data types used by ADM are defined in the `ASTypes.h`, `ADMTypes.h`, `ADMCustomResource.h`, and `ADMAGMTypes.h` files.

**NOTE:** Data types with the AS prefix are used across Adobe products.

To ensure platform independence, ADM and other Adobe products use platform-independent type names for some native data types. The following type definitions are from the `ASTypes.h` file:

### Types defined the same across platforms

```
// Integer Types

typedef signed char ASInt8;
typedef signed short ASInt16;
typedef signed long ASInt32;

typedef unsigned char ASUInt8;
typedef unsigned short ASUInt16;
typedef unsigned long ASUInt32;

typedef long ASErr;

// Storage Types

typedef unsigned char ASByte;
typedef ASByte* ASBytePtr;

// Unicode Types
typedef ASUInt16 ASUnicode;

// Pointer Types

typedef void* ASPtr;
typedef void** ASHandle;
```

```
// Fixed Types

typedef long ASFixed;
typedef long ASFract;
typedef float ASReal;

typedef struct _t_ASFixedPoint {
    ASFixed h, v;
} ASFixedPoint;

typedef struct _t_ASFixedRect {
    ASFixed left, top, right, bottom;
} ASFixedRect;

typedef struct _t_ASFixedMatrix {
    ASFixed a, b, c, d, tx, ty;
} ASFixedMatrix;

typedef struct _t_ASRealPoint {
    ASReal h, v;
} ASRealPoint;

typedef struct _t_ASRealRect {
    ASReal left, top, right, bottom;
} ASRealRect;

typedef struct _t_ASRealMatrix {
    ASReal a, b, c, d, tx, ty;
} ASRealMatrix;
```

## Types defined differently across platforms

```
// Platform structures

// ASBoolean is the same as a Mac OS boolean.
typedef unsigned char ASBoolean;

// ASPortRef is the same as a Mac OS CGrafPtr.
typedef CGrafPtr* ASPortRef;

typedef WindowRef ASWindowRef;

// Rectangle value in Mac OS (same as Rect)
typedef struct Rect ADMRect;

// Point value in Mac OS (same as Point)
typedef struct Point ADMPoint;

// ASBoolean is the same as a Windows BOOL.
typedef int ASBoolean;

// ASPortRef is the same as a Windows HDC.
typedef void* ASPortRef;

// ASWindowRef is the same as a Windows HWND.
typedef void* ASWindowRef;
```

```
// Rectangle value in Windows (same as RECT)
typedef struct _t_ADMRect ADMRect{
   long left, top, right, bottom;
} ADMRect;

// Point value in Windows (same as Point)
typedef struct _t_ADMPoint ADMPoint{
   long h,v;
} ADMPoint;
```

### Coordinates using ADMRect

The `ADMRect` data structure specifies a rectangle of coordinates; however, coordinates are between pixels. For example, using coordinates, if you invalidate (see `sADMDialog->Invalidate`) columns 0 - 3 and columns 4 - 6, the pixels in-between are not re-painted. Below, the periods represent coordinates, and the P's represent pixels. The bold pixels are not re-painted.

. P . P . P . **P** . P . P

. P . P . P . **P** . P . P

## Events

There are five events received by all ADM user-interface objects, as described in the following table.

| Event | When received |
|---|---|
| Destroy | When the object is disposed of |
| Draw | When a screen is invalidated or updated |
| Init | When an object is created |
| Notify | When the object is hit |
| Track | When the mouse is over the object |

For most user-interface objects, you can rely on the default behavior for an event. For instance, when the cursor moves over a text item, ADM changes it to the insert-text cursor.

If the behavior of an object at a given event is not what is desired, it can be changed by assigning a new event handler. One event whose behavior you may modify often is Notify; it is used to check when an object is hit. It is used, for instance, to assign an action to a button click or do special checking on a text-entry item.

## Properties

There are many properties common to all ADM user-interface objects. They are described in the following table. (For more information, see "Glossary.")

| Property | Description |
|---|---|
| **Properties that define the object's function and appearance:** | |
| Type | Defines the general function of the object. Type is the broad category for an object; for example, modal and non-modal dialogs, pop-up menus, and edit-text items. |
| Style | Determines the appearance and/or behavior of the object. The style property further defines the type of object; for example, for dialog objects, it indicates whether a modeless dialog is a tab panel or a standalone window. ADM user-interface objects can have one or more styles. |
| ID | Numeric reference to the object in its defining space (e.g., its resource number or item number). |
| Text | Depending on the item, usually the title, text value, or name of an item. The text associated with an object can be constant, as in a button, or changeable by the user, as with a menu item. |
| **State values:** | |
| Visible | Whether the object is visible. |
| Enabled | Whether the object is enabled. If an object is enabled, it is usable by the user; if disabled, it has a dimmed appearance and may be unusable. |
| Active | Whether the object is the active item, meaning having keyboard focus. There is only one active dialog item object in a given dialog. In Windows®, any item can be active, which for non-text-edit items means it is the focus of the Enter key. In Mac OS®, only edit-text items can be active. |
| Known | Whether the object is known. An item is in a "known" state if it has a "good" or valid value. |
| **Properties that allow ADM objects to access data without the need for global variables:** | |
| Plug-in | A reference to the plug-in that created the object. This is used when the ADM dialog needs to access a plug-in resource. In addition, when dialog elements are created, a pointer to any custom data also is created. This can point to any type of data structure your dialog needs. |
| UserData | A pointer to any special data assigned to the object when it was created. |
| **Properties that define the object's size and location:** | |
| LocalRect | The size of the object. This is the rectangle defining the size of an object in local, (0,0)-based coordinates. |
| BoundsRect | The rectangle of the object in its container's space. A dialog item is located within a dialog, which is located within the screen bounds. This is a rectangle of the same size as LocalRect but in the object container's coordinate space. The coordinates for BoundsRect are measured in screen coordinates, which have the (0,0) origin at the upper left corner of the screen. |

For both LocalRects and BoundsRects, the origin is at the top left of the rectangle, and coordinates increase as they move down and to the right. The origin for tabbed dialogs is beneath the tab, not beneath the window title bar.

## Resources

ADM is designed to simplify the task of creating cross-platform plug-in code for dialogs by largely eliminating the need to support two or more code bases. At the same time, it is intended to support the specific look and feel of its run-time platform. For this reason, dialog resources are created on their host platform, while ADM handles how those dialog resources interact with the user. ADM loads and uses platform-specific dialog resources correctly. The `ADMResource.h` file defines the constants needed when writing ADM dialog resources. The negative IDs are reserved for ADM core implementation, so users should select positive constants for any custom IDs.

In Windows, dialog items are window classes. Variations are controlled by class styles. The mapping of Windows window classes and styles to ADM item types and styles is given in "Windows ADM items" on page 34. Items that take a picture of some sort can use `.bmp` and icon resources. ADM scans for them in that order and uses the first resource it finds with the searched for ID.

In Mac OS, dialogs are made up of normal `'DLOG'` and `'DITL'` resources. Normal dialog item types can be used for standard controls like buttons and text items. Item types unique to ADM are implemented as controls defined in "Mac OS ADM items" on page 38. Items that use pictures of some sort can use PICT and icon family resources to define them. ADM scans for them in that order and uses the first resource it finds with the searched for ID.

Set-up information for ADM objects on all platforms is given in the sections describing specific item types and in "Initializing ADM items" on page 89.

# Suites

The ADM manager provides a set of suites to implement ADM dialogs. The functions of these suites for the current release of ADM are described in Chapter 4, "ADM Suites."

The functions in the suites are standard C style functions. In addition to these, a set of C++ wrappers for working with ADM dialogs as objects is provided in the SDKs for several Adobe products. These wrappers can be found in the `IADM` (Interface to ADM) files in the SDK.

The functions for creating and manipulating ADM user-interface objects are found in several header files. The core suites that make up ADM's public API are shown in the following table.

| Suite | Purpose | Associated header file |
|---|---|---|
| Basic | Provides minimal dialog and resource functions, like alerts, beeps, resource access, and string utilities. | `ADMBasic.h` |
| Dialog | ADM property-access functions for dialog objects. | `ADMDialog.h` |
| Dialog Group | Functions for grouping dialogs into a docked panel. | `ADMDialogGroup.h` |
| Drawer | Functions for implementing custom-drawer callbacks. | `ADMDrawer.h` |
| Entry | Functions for working with ADM Entry objects. | `ADMEntry.h` |
| Hierarchy List | Functions for ADM Hierarchy List objects. | `ADMHierarchyList.h` |
| Icon | Provides a standard interface to cross-platform picture resources. | `ADMIcon.h` |

| Suite | Purpose | Associated header file |
|---|---|---|
| Image | Functions for creating off-screen images that can be displayed and manipulated using ADM drawers. | `ADMImage.h` |
| Item | ADM property-access functions for dialog-item objects. | `ADMItem.h` |
| List | Functions for ADM List objects. | `ADMList.h` |
| List Entry | Functions for ADM Hierarchy List Entry objects. | `ADMListEntry.h` |
| Notifier | Functions for implementing custom-notifier callbacks. | `ADMNotifier.h` |
| Tracker | Functions for implementing custom-tracker callbacks. | `ADMTracker.h` |

Specific API information is provided in the chapters describing each suite.

# ADM object specifics

## ADM Dialog objects

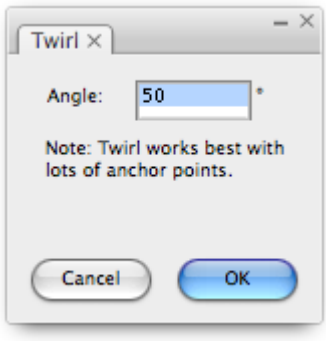
ADM Dialog objects are of two types: modal or non-modal (floating). They are further defined by an ADM dialog style. All ADM Dialog objects have a general appearance that complements the main application's user interface, as shown below.

**Examples of modal dialogs:**

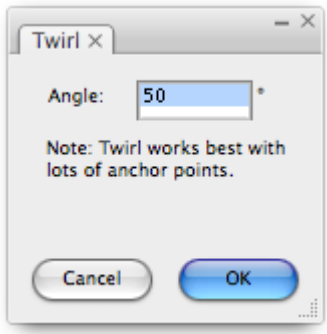

kADMModalDialogStyle


kADMAlertDialogStyle

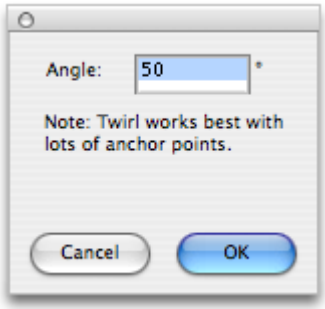
**Examples of floating dialogs:**



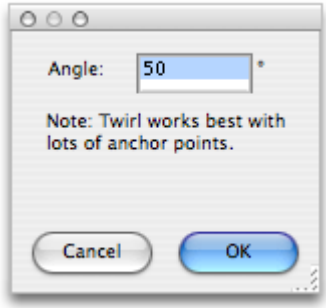

kADMTabbedFloatingDialogStyle




kADMTabbedResizingFloatingDialogStyle



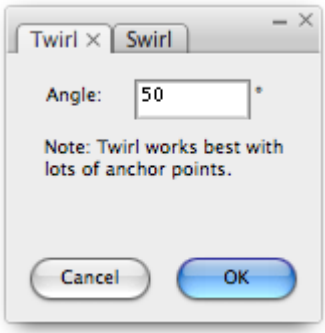

kADMFloatingDialogStyle




kADMResizingFloatingDialogStyle

Modal dialogs require that the user dismiss the dialog before the host application can be directly used again. Often they effect settings or the application's data on being dismissed. For an example of coding a modal dialog, see Chapter 2, "Using ADM with Adobe Photoshop.

Non-modal dialogs, also called floating dialogs or panels, "float" over the main application window and allow the user to switch between the application and the dialog. If a floating dialog is re-sizable, the user can grab the platform-specific resize indicator and stretch or shrink the window area. A floating tabbed dialog can be combined or "docked" with others, as shown below.




For an example of coding a non-modal dialog, see Chapter 3, "Using ADM with Adobe Illustrator.

Behaviors like moving a window or combining several tabbed windows are handled automatically by ADM. ADM handles basic window resizing, but the plug-in probably needs to respond to a resize notification by moving its items or changing their size.

Initially, the size of the window is set by the size of the window resource. It also can be set via a function at any time. In Windows, all ADM dialog windows are standard DIALOG resources. In Mac OS, ADM modeless dialog windows are specified with `'DLOG'` resources using a custom window definition of ID 1991 (WDEF-124 and variation 7). Modal-dialog windows are specified with `'DLOG'` resources using standard Mac OS dialog resources. The following code segments show ADM dialog resources for Windows and Mac OS.

```
/* Mac OS */
resource 'DLOG' (16128) {
       {365, 171, 459, 376},
       1991,
       invisible,
       goAway,
       0x0,
       16128,
       "AlignADM Palette"
};

/* Windows */
16000 DIALOG 12, 9, 161, 67
STYLE WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX |
WS_MAXIMIZEBOX
CAPTION "Align"
FONT 8, "MS Sans Serif"
{
}
```

The ADM window's style is set at runtime when it is created. These styles are found in `ADMDialog.h` and are passed to the dialog creation function.

ADM Dialog objects are created with a plug-in using three calls from the ADM Dialog suite. Two functions, `sADMDialog->Create` and `sADMDialog->Destroy`, are for non-modal (floating) dialogs. To make a non-modal dialog, the plug-in calls `sADMDialog->Create`. When the modeless dialog is no longer needed, the plug-in calls `sADMDialog->Destroy`. For modal dialogs there is only one function, `sADMDialog->Modal`, that is called to create the dialog. Modal dialogs are destroyed automatically when the user dismisses them.

Both `sADMDialog->Create` and `sADMDialog->Modal` take the same arguments:

- `inPluginRef` is for the plug-in creating the dialog.
- `inName` is the name of the dialog window resource. This is an internal name, not the title of the dialog window.
- `inDialogID` is the resource number of the platform dialog resource.
- `inStyle` is one of the ADM dialog style constants in the header files and the examples shown in the first figure in "ADM Dialog objects" on page 18.
- `inInitProc` is a function pointer to a routine that does any initial setup of the dialog, such as positioning it or setting dialog item values.
- The `inData` user argument also is a pointer, but to a structure you define. It is used to access any data needed by the dialog.
- `inOptions` provides additional control on dialog creation.

## Item objects

There are many types of ADM Item objects. Combined with style variations and custom callbacks for drawing, tracking, and notification, you can create just about any dialog appearance and behavior needed. Normally, ADM items associated with a dialog are created automatically with the dialog. You can manually create and dispose of them in your plug-in. Resource types for all ADM items for each platform are given in "Windows ADM items" on page 34 and "Mac OS ADM items" on page 38. "Initializing ADM items" on page 89 explains how to initialize each item.

ADM items are defined in `ADMItem.h`, as is the function suite used to access them. Constants are used to identify each item type. These constants are listed and described below, along with screen shots showing examples of the different item types. In addition to the standard ADM object properties, all ADM items have a parent dialog and a parent-window reference.

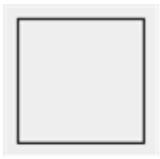
The following table lists ADM item types and the location of short sections that describe them.

| Item type | For more information, see page ... |
|---|---|
| kADMChasingArrowsType | 30 |
| kADMDialType | 30 |
| kADMFrameType | 23 |
| kADMHierarchyListBoxType | 29 |
| kADMItemGroupType | 31 |
| kADMListBoxType | 29 |
| kADMPictureCheckBoxType | 24 |
| kADMPicturePushButtonType | 23 |
| kADMPictureRadioButtonType | 24 |
| kADMPictureStaticType | 23 |

| Item type | For more information, see page ... |
|---|---|
| kADMPopupControlButtonType | 26 |
| kADMPopupControlType | 26 |
| kADMPopupListType | 26 |
| kADMPopupMenuType | 26 |
| kADMPopupSpinEditControlType | 26 |
| kADMProgressBarType | 30 |
| kADMResizeType | 31 |
| kADMScrollbarType | 28 |
| kADMScrollingPopupListType | 26 |
| kADMSliderType | 28 |
| kADMSpinEditPopupType | 28 |
| kADMSpinEditScrollingPopupType | 26 |
| kADMSpinEditType | 28 |
| kADMTextCheckBoxType | 24 |
| kADMTextEditMultilineReadOnlyType | 25 |
| kADMTextEditMultilineType | 25 |
| kADMTextEditPopupType | 26 |
| kADMTextEditReadOnlyType | 25 |
| kADMTextEditScrollingPopUpType | 25 |
| kADMTextEditType | 25 |
| kADMTextMultilineType | 25 |
| kADMTextPushButtonType | 23 |
| kADMTextRadioButtonType | 24 |
| kADMTextStaticType | 25 |
| kADMUserType | 31 |

## kADMFrameType and kADMPictureStaticType

The two simplest ADM Item object types are frames and static pictures, both used primarily for visual effects. ADM frames are used to visually group dialog items together. ADM static pictures are used to provide unchanging visual information to the user, like information about the host program they are using. The following figure shows ADM frames and static pictures.



kADMBlackFrameStyle

kADMSunkenFrameStyle

kADMEtchedFrameStyle

kADMGrayFrameStyle

kADMRaisedFrameStyle


kADMPictureStaticType

The only information needed to define a frame are its bounding rectangle and style. These can be set in the dialog resource or created at runtime. To define a frame in the dialog resource, you would create an item with a specific type of frame style and include a bounds rectangle.

A static picture is defined by its bounding rectangle and a picture resource ID. To define a static picture in the dialog resource, you would provide the resource ID and a bounds rectangle.

## kADMPicturePushButtonType and kADMTextPushButtonType

Buttons are a common dialog control, and ADM offers two types:

- *Text buttons* display the ADM item text within a rounded rectangle.
- *Picture buttons* take three pictures, for their default state, their selected state, and a disabled state.

In addition, a button can be the default item, in which case it is enclosed in another rectangle. In the following figure of ADM button types, `kADMTextPushButtonType` is a default button.

kADMPicturePushButtonType

kADMTextPushButtonType


Enabled Selected Disabled

A text-push button is defined by its bounding rectangle and text. These can be set in the dialog resource or created at runtime. A text-push button is easily defined using a standard platform button-item resource.

A picture-push button is defined by its bounding rectangle and the resource IDs for its three pictures. To define a picture-push button in the dialog resource, use the values in the platform dialog-items chart.

The selected state and disabled state pictures are optional. If resources for these states are not provided, ADM draws them correctly, offsetting the picture when selected and graying it when disabled.

## kADMTextRadioButtonType, kADMPictureRadioButtonType, kADMTextCheckBoxType, and kADMPictureCheckBoxType

Two other types of buttons are made available by ADM:

- *Radio buttons* allow the user to choose one item from a group of options. As with push buttons, radio buttons can be text buttons or picture buttons. They take the same information as push buttons—either the object's text or up to three pictures for the enabled, selected, and disabled states. Radio buttons with consecutive ADM item IDs are automatically grouped together, so only one of the group can be selected.
- *Check boxes* allow the user to set an on/off condition. Check boxes can be of the text or picture type. On Windows, you cannot create a picture check box from platform-specific resources.

Radio buttons and check boxes are illustrated below.


Both check-box and radio-button items have a state that indicates whether they are selected. This can be set by specifying the boolean value of the dialog item:

```
item = sADMDialog->GetItem(parentDialog, kDisableCheckBox);
sADMItem->SetBooleanValue(item, false);
```

Once the value of an item is set, you do not have to set it again unless you choose to do so. ADM's default behavior checks and unchecks a check box or selects and deselects radio buttons in a group. When a radio button in a group is selected, the others in the group are deselected automatically. Radio buttons with consecutive IDs define a button group.

The value of a radio button or check box can be determined by its boolean value:

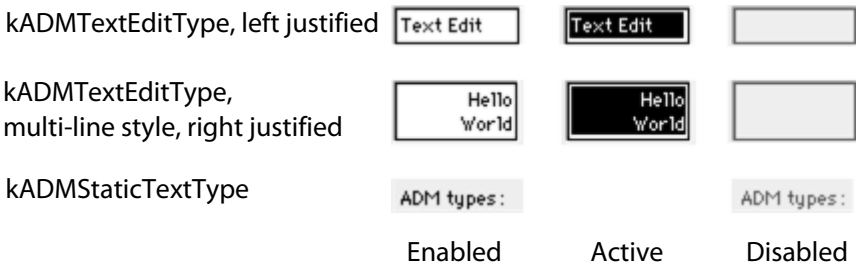
```
item = sADMDialog->GetItem(parentDialog, kDisableCheckBox);
if (sADMItem->GetBooleanValue(item))
       // do something
```

Text-based check boxes and radio buttons can be created by supplying a bounds rectangle and the text to be displayed. To define these text-based items in the dialog resource, use the values in the platform-specific dialog-items chart.

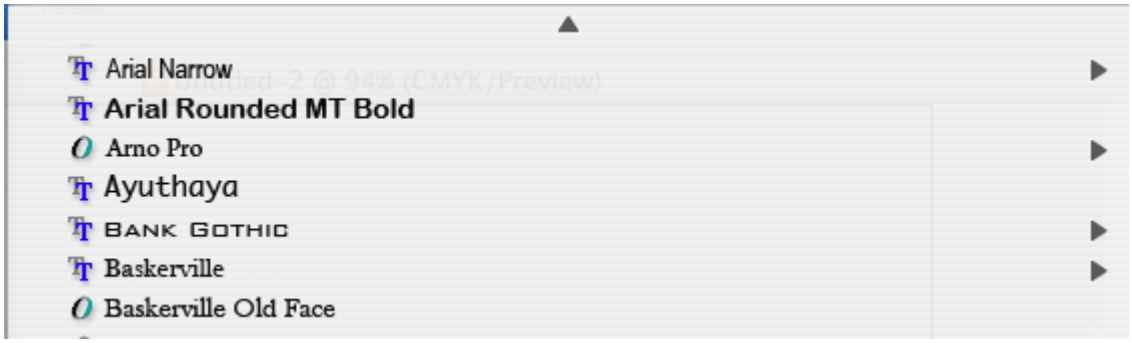
Picture radio buttons are created by supplying a bounds rect and three picture resource IDs. To define a picture push button in the dialog resource, use the values in the platform dialog-items chart. Since only the default picture can be defined in the resource on Windows, the disabled and selected pictures should be defined at runtime when the dialog is initialized.

## kADMTextEditType, kADMTextEditReadOnlyType, kADMTextStaticType, kADMTextEditMultilineType, kADMTextEditMultilineReadOnlyType, and kADMTextEditScrollingPopUpType, kADMTextMultilineType

ADM provides several text items. Text-edit items let the user enter information. Static text items provide information to the user, often as labels for other dialog items. In addition to the two types of text, ADM provides several styles, including numeric items and items with multiple lines. ADM text types and styles are shown below.




An ADM text-edit scrolling pop-up is shown below.




The style of a text field can be set in the dialog resource or at runtime using a constant like the following (see `ADMItem.h`):

```
kADMSingleLineTextEditStyle
kADMNumericTextEditStyle
```

Numeric text fields can have properties that further define the number they can accept, like valid range. See "ADM item numeric properties" on page 32. Multi-line text-edit items display and scroll multiple lines of text, allowing for carriage returns and automatic wrapping as needed.

All text-edit items have a selection range and a maximum length that can be read or set using functions in the ADM Item suite. All text items can have justification set in the dialog resource or at runtime using one of these constants:

```
kADMLeftJustify
kADMCenterJustify
kADMRightJustify
```

Numeric text items can have a units value automatically appended to the text. A text field can have one of the following units:

```
kADMNoUnits
kADMPointUnits
kADMInchUnits
kADMMillimeterUnits
kADMCentimeterUnits
kADMPicaUnits
kADMPercentUnits
kADMDegreeUnits
```

No text is appended to a numeric text item if its units property is `kADMNoUnits`. The units to use can be set at runtime using ADM text-item functions.

Static text items often are used as labels for items. A standard behavior for static text labels for text-edit items is for the text item to become active when the label is selected. ADM automatically provides this behavior if the static text label ID immediately precedes or follows the edit-text ID.

The text of *any* ADM item can be set and retrieved using two text-item functions:

```
char text[65];
item = sADMDialog->GetItem(parentDialog, kSomeTextItem);
sADMItem->GetText(item, text, 65);
updateText(text);
sADMItem->SetText(item, text);
```

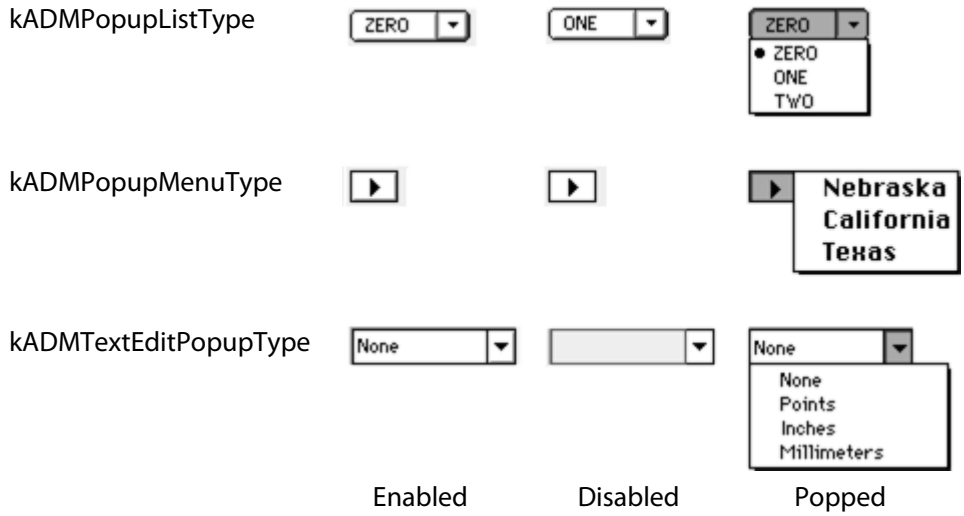
Text items are defined by their bounds rectangle, style, justification, and some text. Their bounds and justification can be set in the dialog resource, as indicated by the dialog-item resource tables. Other properties of a text item are specified at runtime when the dialog is initialized.

**NOTE:** The read-only versions of text edit items do not have a platform-specific component.

## kADMPopupListType, kADMPopupMenuType, kADMScrollingPopupListType, kADMTextEditPopupType, kADMPopupControlType, kADMPopupControlButtonType, kADMPopupSpinEditControlType, and kADMSpinEditScrollingPopupType

Pop-up items are a common user-interface item in dialogs, and ADM provides several variations on the basic pop-up. Pop-up menus and lists allow the user to choose one item from a list of options that becomes visible when the item is selected. Generally, pop-up menus and lists are text only, with a standard platform menu resource defining the list of options for the user.

Pop-up list items display their current setting to the user. Pop-up menu items appear when the item is selected and you would likely act immediately on the user's selection or display it elsewhere. An ADM text-edit pop-up menu is a combination of a text-edit field as described above and a pop-up menu. The user's pop-up menu selection is placed in the text edit field. ADM pop-up items are illustrated below.




An ADM pop-up menu can be one of 10 styles, 2 of which are shown below. The style variant determines where the pop-up menu appears. ADM uses this item to create certain item types (e.g., the window menu discussed below and text-edit pop-up items). While you can use pop-up menus, pop-up lists are more common.

```
kADMRightPopupMenuStyle
kADMBottomPopupMenuStyle
```

A common use for pop-up menus within ADM is to place a menu to the right of tabs in a floating tabbed window. It is made visible when entries are added to it. Because the origin for tabbed dialogs is beneath the tab and not beneath the window title bar, menu items of this sort have a bounds rectangle with a negative top and 0 for its bottom. You do not need to create this item; ADM creates it automatically for tab-style windows. Its item ID is `kADMMenuItemID`.

A pop-up text-edit item can be one of the four styles listed below:

```
kADMSingleLineEditPopupStyle
kADMExclusiveEditPopupStyle
kADMNumericEditPopupStyle
kADMDummyTextEditPopupStyle
```

If you want to manipulate individual items in a menu, the ADM menu item is treated as an ADM List object. The list reference for a menu item is obtained using the `sADMItem->GetList` function. There is a suite of functions for performing list operations—the ADM List suite. The items in a menu actually are ADM objects called ADM entries. ADM Entry objects can be enabled or active, like any other ADM object. They also can be checked to indicate the current menu value. ADM List objects and ADM Entry objects are discussed more later.

The value of a pop-up item (the position of its selected item) can be retrieved using the ADM List and ADM Entry suites. You get the active entry in the list, then get the index of the entry. See below.

```
ADMItemRef item = sADMDialog->GetItem(parentDialog, kSomeMenuItem);
ADMListRef list = sADMItem->GetList(item);
ADMEntryRef entry = sADMList->GetActiveEntry(list);
ASInt32 selection = sADMEntry->GetIndex(entry);
```

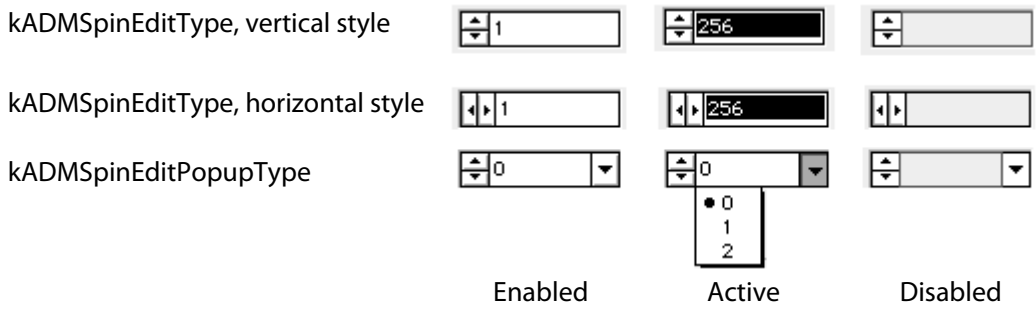
To get the name, use the `sADMEntry->GetText` function instead.

Menu items are defined by their bounds rectangle, a menu resource ID, and a style. The bounds rectangle and style of pop-up items are defined in a resource, as indicated in the item-resource tables. The menu

resource ID of the pop-up menu's list is specified at runtime when the dialog is initialized. In both Mac OS and Windows, the menu resource type is `'MENU'`.

## kADMSpinEditType and kADMSpinEditPopupType

A variation of a text-edit item is a spin-edit item. Spin-edit items provide arrows to increase and decrease their value without typing. A further variation is a spin-edit pop-up item, which adds a pop-up menu to the spin-edit item. Spin-edit items have many of the same properties as text-edit items, like justification. They are inherently numeric items and have those properties as well. ADM spin-edit items are illustrated below.




Spin-edit items can have horizontal or vertical arrows, as specified by their style:

```
kADMVerticalSpinEditStyle
kADMHorizontalSpinEditStyle
```

Spin-edit pop-up items also may have a vertical or horizontal style:

```
kADMVerticalSpinEditPopupStyle
kADMHorizontalSpinEditPopupStyle
```

The rate at which a spin-edit control changes the number in its edit field is controlled by its small-increment value.

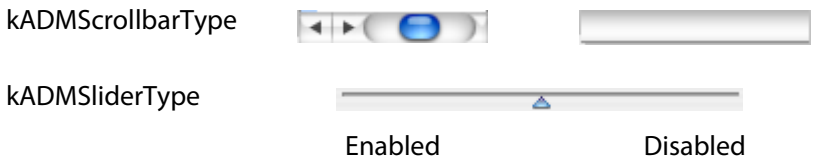
The value of a spin-edit item can be retrieved in one of two ways: you can get its value or its text. To get the text, use `sADMDialog->GetText`. To get the value, use the appropriate "get value" function on the spin item. For instance, to get an integer value, you would use the following:

```
ASInt32 selection;
item = sADMDialog->GetItem(parentDialog, kSomeMenuItem);
selection = sADMItem->GetIntValue(item);
```

A spin-edit item is defined by a bounds rectangle and a style, which can be defined in the dialog-item resource. When the dialog is initialized, the other properties of the item, like its value and justification, can be defined.

## kADMScrollbarType and kADMSliderType

Scrollbars and sliders allow the user to select from a range of values with a graphic interface. The relative position of the current value within the range is indicated by the position of the item's "thumb"— the triangle on the slider and the rectangle within the scrollbar. The item's value can be changed by dragging the thumb. A scrollbar's value also can be changed, using the arrows at its ends. Scrollbars and sliders are illustrated below.

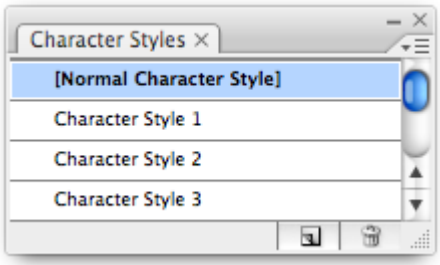



The rate at which a scrollbar item changes its value is controlled by its large- and small-increment values. The small increment is used when the arrows are clicked; the large increment, when the user clicks inside the scrollbar. The value of a slider or scrollbar item can be retrieved using the appropriate "get value" function on the item. See "ADM item numeric properties" on page 32.

Scrollbar and slider items are defined by their bounds rectangle and a range. The bounds rectangle is specified in the dialog-item resource. The other properties, including range and large and small increments, are defined at runtime when the dialog is initialized.

## kADMListBoxType and kADMHierarchyListBoxType

List boxes display a list of options and allow the user to select one or more of them. Their current selection is indicated to the user by inverting the items. If more items are in the list than can be displayed, a scroll bar allows the user to navigate the list. While lists often are text only, they may include graphical information like a color preview or icon. The display of pictures is handled automatically. More complex lists are created by overriding the list drawing routine. A list item with New and Delete buttons is shown below.




ADM provides several variations on the basic text-item list. A list box can allow only one item to be selected or allow multiple items to be selected. A list also can be created with or without dividing lines between objects. These options are expressed using flags that are ORed together:

```
/* List box styles */
#define kADMMultiSelectListBoxStyle                (1L<<0)
#define kADMDividedListBoxStyle                    (1L<<2)
#define kADMEntryTextEditableListBoxStyle          (1L<<3)
```

Some combinations of these ADMListBox style options are as follows:

```
#define kADMSingleSelectListBoxStyle               0
#define kADMMultiSelectListBoxStyle                (kADMMultiSelectListBoxStyle)
#define kADMMultiSelectTileListBoxStyle
          (kADMMultiSelectListBoxStyle|kADMTileListBoxStyle)
#define kADMSingleSelectDividedListBoxStyle        (kADMDividedListBoxStyle)
#define kADMMultiSelectDividedListBoxStyle
          (kADMMultiSelectListBoxStyle|kADMDividedListBoxStyle)
#define kADMMultiSelectTileDividedListBoxStyle
          (kADMMultiSelectListBoxStyle|kADMTileListBoxStyle|kADMDividedListBoxStyle)
```

The list item actually is a container object for a list and its entries. Each item in a list is an ADM object called an ADM Entry object. ADM Entry objects can be enabled or selected, like any other ADM object. They can have special draw functions for custom displays. ADM entry items also are used by ADM pop-up items. Also available is a suite of functions for performing list operations like controlling a list's appearance and indexing through its entries.

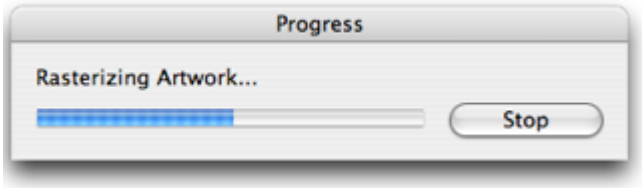
How you retrieve the list selection depends on the list style. In general, you get the selected entry reference or references, then use the reference to obtain specific information. The following code shows how you might get the selection values from a multi-selection list.

```
ASInt32 selectedCount = sADMList->NumberOfSelectedEntries(theList);
for (i = 0; i < selectedCount; i++) {
    ADMEntryRef theEntry = sADMList->IndexSelectedEntry(theList, i);
    // do something to the entry
    ASInt32 index = sADMEntry->GetIndex(theEntry);
    ...
}
```

List-box items are defined by their bounds rectangle and a style. The bounds rectangle and style of pop-up items are defined in a resource, as indicated in the item-resource tables. Lists can be filled automatically by assigning a menu ID at runtime. Other initialization also is done at runtime when the dialog is initialized.

## kADMProgressBarType

An ADM progress bar indicates a lengthy operation is occurring. This item uses a `CNTL` resource in Mac OS and can be created programmatically in Windows. A progress bar is shown below.




## kADMChasingArrowsType

ADM "chasing arrows" indicate through a simple animation that a background process is in progress. This is available only in Mac OS and is shown below.

## kADMDialType

An ADM dial is used for calibration. To initialize, you must set its initial value, maximum value, and minimum value. A dial is shown below.

### kADMItemGroupType

An ADM item group is a collection of individual items. Item groups make it easier to write notification and tracker callbacks, since multiple items are dealt with as though they are one. Item groups have no physical representation; they are simply an organizational grouping, hence are not defined by any specific platform resource. All items respond to single function calls to the group. See "ADM item groups" on page 31.

### kADMUserType

ADM User and ADM Custom items are used indirectly together to extend ADM with completely new items. A plug-in that provides a custom ADM item uses an item of type `kADMUserType` as a foundation and customizes its behavior. Custom items are discussed further in "Adding custom item types" on page 45.

### kADMResizeType

This item is created automatically by ADM when a resizable dialog is created and displays the platform's window resize item. Notification of a window being resized occurs if a notifier handler function is assigned to this item. This function handles resizing or repositioning items in the dialog. For more information, see "Resizable windows" on page 45.

## Composite items

Some ADM items actually are two or more ADM items composited together. These are list items, spin-edit items, spin-edit pop-up items, and text-edit pop-up items. The normal ADM item reference to such an item is to the composite object. The components, or children, of the item can be accessed and then used like any other ADM item; for instance, setting a custom-notifier callback function.

The children of a composite item are accessed using the `sADMItem->GetChildItem` function, which is passed a `ChildID` argument. The `ChildIDs` for each composite item are defined in the `ADMItem.h` file. For instance, a list item has these children:

```
typedef enum
{
    kADMListBoxScrollbarChildID = 1,
    kADMListBoxListChildID = 2,
    kADMListBoxTextEditBoxChildID = 3,
    kADMListBoxPopupChildID = 4,
    kADMListBoxDummyChildID = 0xFFFFFFFF
}
ADMListBoxChildID;
```

## ADM item groups

If a composite item is not available, an ADM item group allows you to collect together several items that need to respond to calls as a group. For example, you might have five items that need to be enabled or disabled simultaneously. Once those items belong to a group, you need to enable/disable just the group.

This is not true of geometrical containment. Item groups really do not have any physical manifestation; they are simply a way of logically grouping items.

## ADM item numeric properties

ADM items often have a numeric value. Four properties can be used to control this value, providing automatic bounds checking or feedback:

- **`type`** — The numeric `type` refers to how the value is set and retrieved. The valid types are boolean, integer, fixed, and float. Not all item types have these numeric types; for instance, a check box has only a boolean value, while a slider can have any of them. Values are accessed using get and set functions for the type of data desired; for instance, `sADMItem->GetFixedValue` or `sADMItem->GetMinIntValue`. The data type of an item is typecast by the function used to access it. For instance, if the boolean value of a check box is retrieved with `sADMItem->GetFloatValue`, it is returned as `0.0` or `1.0`.

- **`precision`** — The `precision` property of an item refers to how many digits follow the decimal point. Values of an item are automatically limited to the defined precision.

- **`range`** — All items except boolean items can have an assigned `range` that sets upper and lower limits on the values that can be assigned to it. ADM automatically confines the value to this specified range in one of two ways. For text-edit and spin-edit items, a note alert appears, informing the user of the valid range if an illegal value is entered. The value is then floored or ceilinged to bring it into range. For sliders and scrollbars, the range is used to calibrate the dialog item. The minimum range value corresponds to the item value when the thumb is in the leftmost position; the maximum when the thumb is in the rightmost position. The range values are accessed using get and set functions for the type of data desired; for instance, `sADMItem->GetMinIntValue` or `sADMItem->SetMaxFloatValue`.

- **`increment`** — You can set the rates at which scrollbar and spinner item values change by setting their `increment` properties. There are small and large increments. The small value is added to or subtracted from the value when an arrow component of the item is clicked once. The large increment is used only by scrollbar items and is added to or subtracted from the item value when the user clicks above or below the thumb inside the scrollbar. The increment values are accessed using get and set functions for the size of increment; for instance, `sADMItem->SetSmallIncrement`.

**NOTE:** Increments are floats and always are in the specified units for an item.

For information on text-to-float and float-to-text conversions, see "FloatToText and TextToFloat functions" on page 92.

## ADM List and ADM Entry objects

ADM items based on a list of choices include list boxes, pop-up lists, pop-up menus, spin-edit popups, and text-edit popups. All are accessed in the same way, as lists of entries. There are two suites of functions used to access the list and entry objects, the ADM List suite and ADM Entry suite. The ADM List suite basically lets you access ADM entries. With it, you can add and remove entries, iterate through the existing ones, and control the list's entries' height and width. Once you have used the ADM List suite to access an individual entry, you can use the ADM Entry suite to modify its properties. ADM entries are similar to other ADM user-interface objects, having properties like ID and text and states like enabled and active.

Entries do not have these standard properties: plug-in, type, style, and visible state. They have these additional properties:

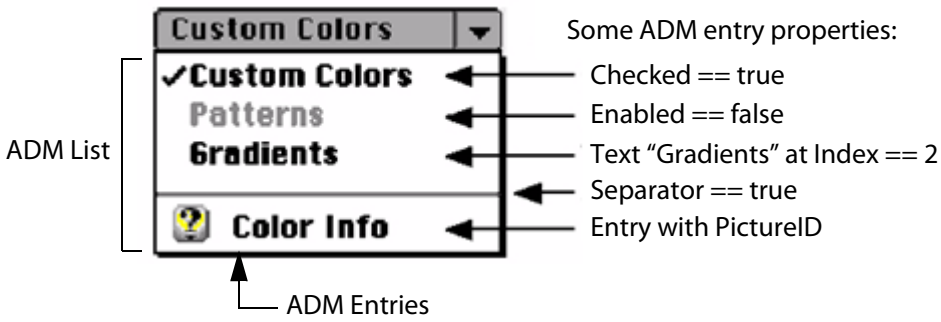
- The *index* is the position of the entry in the list.

- The *selected state* indicates the user selected the item (others may be selected in the case of a multi-select list).

- The *checked state* indicates a check mark appears to the left of the entry.
- The *separator state* indicates the item is a non-selectable item used to break a list into groups of entries.

If an entry has an assigned picture, it is automatically drawn to the left of the text. In addition, an entry's event-handler routines cannot be overridden. Special event handling is done by the parent list.

ADM list and entry objects are illustrated below.




To get the list object for an item, use `sADMItem->GetList`. Once this is done you can use the ADM List and ADM Entry suites' functions to modify it.

An item's list can be initialized by repeatedly creating entries with `sADMList->InsertEntry` and then using `sADMEntry->SetText` to set the new entry's text:

```
for (index = 0; index < kNumberEntries; index++) {
       char menuText[255];
       ADMEntryRef entry = sADMList->InsertEntry(theItemList, index);
       sBasic->GetIndexString(thePlugin, 16000, index, menuText, 255);
       sADMEntry->SetText(entry, menuText);
}
```

or more quickly by assigning it a menu resource ID:

```
sADMList->SetMenuID(theItemsList, gPlugInRef, 16000, "Choices");
```

In this case, the list items are set corresponding to the items already created in the resource.

Iterating through a list's items is done in a similar fashion to the example given under the `kADMListItem` description.

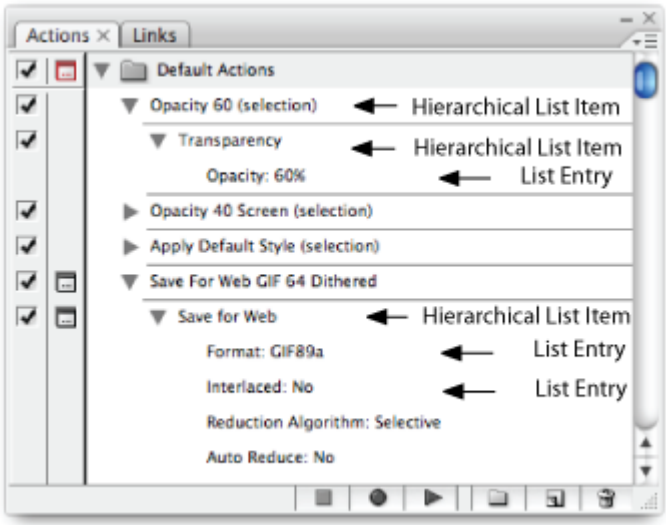
**NOTE:** List indices are 0-based.

## ADM Hierarchy List and ADM List Entry objects

The ADM Hierarchy List suite allows you to access ADM Hierarchy List objects and ADM List Entry objects. Since an ADM Hierarchy List object is an extended property of a standard ADM Item object, this suite lacks many of the functions common to ADM objects; however, you can access the hierarchy list's ADM item and do common operations on it. Using functions in this suite, you can initialize the hierarchy list, and you can create, destroy, customize, and iterate through the ADM list entries of a hierarchy list. The Hierarchy List suite is used in conjunction with the ADM List Entry suite to further access list-related information.

**NOTE:** The relationship between ADM Hierarchy List objects and ADM List Entry objects is the same as that between ADM List objects and ADM Entry objects; that is, list entries are the elements of a hierarchy list. List entries themselves may be hierarchy lists with list entry children of their own.

ADM hierarchy list and list-entry objects are illustrated below.




**NOTE:** List indices are 0-based.

## Windows and Mac OS ADM item-resource lists

The tables in "Windows ADM items" on page 34 and "Mac OS ADM items" on page 38 list the resource information needed to define ADM items in their native resource formats.

Windows ADM items are defined by the dialog-item window class and style which map to an ADM item and style. The item values are set at runtime using the ADM Item suite functions.

**NOTE:** In the resource file, set the item name to the picture ID to use.

In Mac OS, ADM items are created using a dialog-item list resource (`DITL`). Simple text-based items like text push buttons and text-edit, can be made using standard Mac OS dialog items. Others are indicated using control items (`CNTL`) with the appropriate `CDEF` and variation (or ProcID, which CDEF * 16 + variation). The variation and other values can also be set at runtime.

**NOTES:** (1) In Windows plug-ins, these values cannot be set in the resource but must be set at runtime. Because of this, you may want to set them at runtime on both platforms.
(2) On Windows, this can be set in a resource only for single-line text-edit items.

### Windows ADM items

| ADM item type | ADM style | Windows class name | Window style |
|---|---|---|---|
| ADM Custom | — | *custom item name* | "" |
| ADM Dial | — | "ADM Dial Type" | 0 |

| ADM item type | ADM style | Windows class name | Window style |
|---|---|---|---|
| ADM Frame | `kADMSunkenFrameStyle` | "Button" | BS_GROUPBOX |
| | `kADMBlackFrameStyle` | "Static" | SS_BLACKRECT |
| | `kADMGrayFrameStyle` | "Static" | SS_GRAYRECT |
| | `kADMRaisedFrameStyle` | "Static" | SS_WHITERECT |
| | `kADMBlackFrameStyle` | "Static" | SS_BLACKFRAME |
| | `kADMGrayFrameStyle` | "Static" | SS_GRAYFRAME |
| | `kADMRaisedFrameStyle` | "Static" | SS_WHITEFRAME |
| | `kADMSunkenFrameStyle` | "Static" | SS_ETCHEDHORZ |
| | `kADMSunkenFrameStyle` | "Static" | SS_ETCHEDVERT |
| | `kADMSunkenFrameStyle` | "Static" | SS_ETCHEDFRAME |
| | `ADMFrameStyle` | "ADM Frame Type" | — |
| ADM List Box | — | "Listbox" | — |
| | `ADMListBoxStyle` | "ADM List Box Type" | — |
| | `ADMListBoxStyle` | "ADM Hierarchy List Box Type" | — |
| ADM Password Text Edit | — | "ADM Text Edit Type" | ES_PASSWORD |
| ADM Picture Push Button (0) | `item name == MAKEINTRESOURCE(pictureID)` | "ADM Picture Push Button Type" | 0 |
| ADM Picture Radio Button (0) | `item name == MAKEINTRESOURCE(pictureID)` | "ADM Picture Radio Button Type" | 0 |
| ADM Picture Static (0) | `item name == MAKEINTRESOURCE(pictureID)` | "Static" | SS_BITMAP |
| | `item name == MAKEINTRESOURCE(pictureID)` | "Static" | SS_ICON |
| | `item name == MAKEINTRESOURCE(pictureID)` | "Static" | SS_ENHMETAFILE |
| | `item name == MAKEINTRESOURCE(pictureID)` | "ADM Picture Static Type" | 0 |
| ADM Popup List | — | "Combobox" | CBS_DROPDOWNLIST |
| | — | "ADM Popup List Type" | 0 |
| ADM Popup Control | — | "ADM Popup Control Type" | 0 |
| ADM Popup Control Button | — | "ADM Popup Control Button Type" | 0 |

| ADM item type | ADM style | Windows class name | Window style |
|---|---|---|---|
| ADM Popup Spin Edit Control | — | "ADM Popup Spin Edit Control Type" | 0 |
| ADM Popup Menu | `ADMPopupMenuStyle` | "ADM Popup Menu Type" | 0 |
| ADM Resize | — | "ADM Resize Type" | 0 |
| ADM Scrollbar | — | "Scrollbar" | 0 |
| | — | "ADM Scrollbar Type" | 0 |
| ADM Scrolling Popup List | — | "Combobox" | CBS_DROPDOWNLIST\| WS_VSCROLL |
| | — | "ADM Scrolling Popup List Type" | 0 |
| ADM Slider | `ADMSliderStyle` | "MSCtls_Trackbar32" | 0 |
| | — | "ADM Slider Type" | 0 |
| ADM Spin Edit | `ADMSpinEditStyle` | "ADM Spin Edit Type" | — |
| ADM Spin Edit Popup | `kADMSingleLineEditPopupStyle` | "Combobox" | CBS_DROPDOWN |
| | `ADMSpinEditPopupStyle` | "ADM Spin Edit Popup Type" | — |
| ADM Spin Edit Scrolling Popup | `kADMSingleLineEditPopupStyle` | "Combobox" | CBS_DROPDOWN\| WS_VSCROLL |
| | `ADMSpinEditPopupStyle` | "ADM Spin Edit Scrolling Popup Type" | — |
| ADM Text Check Box | — | "Button" | BS_CHECKBOX |
| | — | "Button" | BS_AUTOCHECKBOX |
| | — | "Button" | BS_3STATE |
| | — | "Button" | BS_AUTO3STATE |
| | — | "ADM Text Check Box Type" | 0 |
| ADM Text Edit | `kADMLeftJustify` | "Edit" | ES_LEFT |
| | `kADMCenterJustify` | "Edit" | ES_CENTER |
| | `kADMRightJustify` | "Edit" | ES_RIGHT |
| | `kADMNumericTextEditStyle` (Auto sets if you call `SetXValue`) | "Edit" | ES_NUMBER |
| | `ADMTextEditStyle` | "ADM Text Edit Type" | — |
| | `kADMPasswordTextEditStyle` | "Edit" | ES_PASSWORD |

| ADM item type | ADM style | Windows class name | Window style |
|---|---|---|---|
| ADM Text Edit Multi Line | `kADMNumericTextEditStyle` (Auto sets if you call `SetXValue`) | "Edit" | ES_MULTILINE |
| | — | "ADM Text Edit Multi Line Type" | 0 |
| ADM Text Edit Popup | `kADMSingleLineEditPopupStyle` | "Combobox" | CBS_DROPDOWN |
| | `ADMTextEditPopupStyle` | "ADM Text Edit Popup Type" | |
| ADM Text Edit Scrolling Popup | `kADMSingleLineEditPopupStyle` | "Combobox" | CBS_DROPDOWN\| WS_VSCROLL |
| | `ADMTextEditPopupStyle` | "ADM Text Edit Scrolling Popup Type" | — |
| ADM Text Push Button | Default | "Button" | BS_DEFPUSHBUTTON |
| | — | "Button" | BS_PUSHBUTTON |
| | — | "Button" | BS_USERBUTTON |
| | — | "Button" | BS_OWNERDRAW |
| | — | "ADM Text Push Button Type" | 0 |
| ADM Text Radio Button | `ADMRadioButtonStyle` | "Button" | BS_RADIOBUTTON |
| | — | "Button" | BS_AUTORADIOBUTTON |
| | — | "ADM Text Radio Button Type" | 0 |
| ADM Text Static | `kADMLeftJustify` | "Static" | SS_LEFT |
| | `kADMCenterJustify` | "Static" | SS_CENTER |
| | `kADMRightJustify` | "Static" | SS_RIGHT |
| | `kADMLeftJustify` | "Static" | SS_LEFTNOWORDWRAP |
| | `kADMLeftJustify` | "Static" | SS_SIMPLE |
| | — | "ADM Text Static Type" | 0 |
| ADM Text Static Multi Line | — | "Edit" | ES_READONLY |
| | — | "ADM Text Static Multi Line Type" | 0 |
| ADM User | — | "ADM User Type" | 0 |

## Mac OS ADM items

| | | Mac CNTL resource settings | | Additional fields | | | |
|---|---|---|---|---|---|---|---|
| **ADMItem type** | **Mac dialog item** | **CDEF res ID** | **Variation** | **Value** | **Min** | **Max** | **Other** |
| ADM Chasing Arrows | Control Item | 7 | 0 | | | | |
| ADM Custom | Control Item | 1090 | CNTL Title = "Name Registered Custom Item Type" | | | | |
| ADM Dial | Control Item | 1045 | 0 | IntValue (1) | IntMin (1) | IntMax (1) | |
| ADM Frame | User Item, Control Item | 1000 | ADMFrameStyle | | | | Control title is the group name (1) |
| ADM Hierarchical List | Control Item | 1011 | 0 | | | | |
| ADM List Box | Control Item | 1010 | ADMListBoxStyle | | MenuID= MenuResID (1) | | |
| ADM Picture Check Box | Control Item | 1023 | ADMPictureButton Style | PictureID | Picture SelectedID (1) | Picture DisabledID (1) | |
| ADM Picture Push Button | Control Item | 1020 | ADMPictureButton Style | PictureID | Picture SelectedID (1) | Picture DisabledID (1) | |
| ADM Picture Radio Button | Control Item | 1021 | ADMPictureButton Style | PictureID | Picture SelectedID (1) | Picture DisabledID (1) | |
| ADM Picture Static | Icon Item, Picture Item, Control Item | 1022 | 0 | PictureID | Picture SelectedID (1) | Picture DisabledID (1) | |
| ADM Popup Control | Control Item | 1055 | 0 | ADM Justify (1)<br>IntValue (1) | IntMin (1) | IntMax (1) | |

| | | Mac CNTL resource settings | | Additional fields | | | |
|---|---|---|---|---|---|---|---|
| **ADMItem type** | **Mac dialog item** | **CDEF res ID** | **Variation** | **Value** | **Min** | **Max** | **Other** |
| ADM Popup Control Button | Control Item | 1056 | 0 | ADM Justify (1)<br>IntValue (1) | IntMin (1) | IntMax (1) | |
| ADM Popup List | Control Item | 63 | 0 | | MenuID= MenuResID (1) | | |
| ADM Popup Menu | Control Item | 1030 | ADMPopupMenu Style | | MenuID= MenuResID (1) | | |
| ADM Popup Spin Edit Control | Control Item | 1057 | 0 | ADM Justify (1)<br>IntValue (1) | IntMin (1) | IntMax (1) | |
| ADM Progress Bar | Control Item | 5 | 0 | IntValue (1) | IntMin (1) | IntMax (1) | |
| ADM Resize | Control Item | 1040 | 0 | | | | |
| ADM Scrollbar | Control Item | 1 | 0 | IntValue (1) | IntMin (1) | IntMax (1) | |
| ADM Scrolling Popup List | Control Item | 1031 | 0 | | | | |
| ADM Slider | Control Item | 1050 | 0 | IntValue (1) | IntMin (1) | IntMax (1) | |
| ADM Spin Edit | Control Item | 1060 | ADMSpinEditStyle | ADM Justify (1) | | | |
| ADM Spin Edit Popup | Control Item | 1061 | ADMSpinEditPopup Style | ADM Justify (1) | MenuID= MenuResID (1) | | |
| ADM Spin Edit Scrolling Popup | Control Item | 1062 | ADMSpinEditPopup Style | ADM Justify (1) | MenuID= MenuResID (1) | | |
| ADM Tabbed Menu | Deprecated —do not use | — | — | — | — | — | — |
| ADM Text Edit | Text-edit Item, Control Item | 1070 | ADMTextEditStyle | ADM Justify (2) | | | |

| | | Mac CNTL resource settings | | Additional fields | | | |
|---|---|---|---|---|---|---|---|
| **ADMItem type** | **Mac dialog item** | **CDEF res ID** | **Variation** | **Value** | **Min** | **Max** | **Other** |
| ADM Text Check Box | Check Box Item<br>Control Item | 0 | 1 | | | | |
| ADM Text Edit Multi Line | Control Item | 1073 | 0 | ADM Justify (1) | | | |
| ADM Text Edit Popup | Control Item | 1071 | ADMTextEditPopup Style | ADM Justify (1) | | MenuID=Menu ResID (1) | |
| ADM Text Edit Scrolling Popup | Control Item | 1075 | ADMTextEditPopup Style | ADM Justify (1) | | MenuID=Menu ResID (1) | |
| ADM Text Push Button | Push Button Item, Control Item | 0 | 0 | | | | |
| | Control Item | 0 | 4=Default | | | | itemID = 1 is made default automatically |
| ADM Text Radio Button | Radio Button Item, Control Item | 0 | 2 | | | | |
| ADM Text Static | Static Text Item, Control Item | 1072 | 0 | ADM Justify (1) | | | |
| ADM Text Static Multi Line | Control Item | 1074 | 0 | ADM Justify (1) | | | |
| ADM User | Control Item | 1080 | 0 | | | | |
| ADM Unicode Text Edit | Control item | 1100 | 0 | | | | |
| ADM Password Text Edit | Control item | 1200 | 0 | | | | |

# Using event callbacks

In general, each ADM object has a default function for each event. If you need only the normal behavior of an item, you can ignore its events and handler functions and rely on the defaults.

If you want a custom behavior for an item, its standard handler functions can be replaced by custom handler functions. The new handler function will likely call the default function and supplement its behavior. Custom handlers for draw, track, and notify events are called Drawers, Trackers, and Notifiers, respectively. Custom init and destroy functions are implemented using standard C and API functions. ADM Tracker, ADM Drawer, and ADM Notifier functions also can use their related suites. Events are received by all objects in the object-container hierarchy. For instance, if the object is an ADM button item, the ADM item receives the event followed by its containing ADM dialog.

Replacement of a dialog or item event handler function is done using definitions and functions in the ADM suite for the object type. ADM Entry and ADM List Entry objects can have custom handler functions, but these are set by their parent list. ADM List objects are handled by their parent dialog item.

If the handler can be changed, there is a `Set`*`Event`*`Proc` function for the handler in the object suite. If the default handler function can be called, there is a `Default`*`Event`* function in the suite. For instance, to override the default drawing behavior of an ADM Item object, use these definitions and functions:

```
typedef void ASAPI (*ADMItemDrawProc)(ADMItemRef inItem,
    ADMDrawerRef inDrawer);

void ASAPI (*SetDrawProc)(ADMItemRef inItem, ADMItemDrawProc inDrawProc);
void ASAPI (*DefaultDraw)(ADMItemRef inItem, ADMDrawerRef inDrawer);
```

The new handler function must follow the correct function prototype, which is defined to have enough information to handle the event. For instance, your custom draw-item function would receive the item to draw and a drawer reference used to draw the item.

ADM objects and events that can have custom handlers are listed in the following table.

| Event/customization | ADM dialog | ADM item | ADM entry |
|---|---|---|---|
| Init | | | |
| Customizable | Y | Y | Y |
| Can Call Default | Y | Y | N (see note below table) |
| Draw | | | |
| Customizable | Y | Y | Y |
| Can Call Default | Y | Y | Y |
| Track | | | |
| Customizable | Y | Y | Y |
| Can Call Default | Y | Y | Y |
| Notify | | | |
| Customizable | Y | Y | Y |
| Can Call Default | Y | Y | Y |
| Destroy | | | |
| Customizable | Y | Y | Y |
| Can Call Default | Y | Y | N (see note below table) |

**NOTE:** Custom ADM entry functions are set by the parent list.

**NOTE:** ADM entry item create and destroy functions are called from the parent list object, unlike the draw, track and notify functions, which can be called by the item handler.

## Init functions

ADM initialization functions for dialogs and items are passed in when the `sADMDialog->Create` function is called. ADM lists do not have a unique init function but are treated as ADM items. The only ADM object to which you assign a new init routine is an ADM entry, and this actually is assigned to the parent ADM List object. Each time an entry is added to the list, the initialization function is called.

The general format of init functions is given below. When the init function is called for an object, a reference to the new object is passed to it:

```
typedef ADMErr ADMAPI (*ADMObjectInitProc)(ADMObjectRef inObject);
```

The example below shows two init functions, for a dialog and an entry. The dialog init function actually sets the entry init function and item-handler functions:

```
ADMErr myDialogInit(ADMDialogRef dialog) {
    ADMItemRef initItem;
    ADMListRef myList;

    initItem = sADMDialog->GetItem(myDialog, kOKButton);
    sADMItem->SetNotifyProc(initItem, myOKHandler);

    initItem = sADMDialog->GetItem(myDialog, kList);
    myList = sADMItem->GetList(initItem);
    sADMList->SetInitProc(myList, myListEntryInit);

    initItem = sADMDialog->GetItem(myDialog, kCustomItem);
    sADMItem->SetDrawProc(initItem, mySquareDrawHandler);
    sADMItem->SetTrackProc(initItem, mySquareTrackHandler);
}

ASErr myListEntryInit(ADMEntryRef entry) {
    // init stuff such as setting a color or a pointer
}
```

The dialog init function is passed to the dialog when it is created with the `sADMDialog->Create` or `sADMDialog->Modal` functions:

```
ADMInt32 dialogDismissButton = sADMDialog->Modal(gPlugInRef, "DialogName",
    kMyDialogID, kADMModalDialogStyle, myDialogInit, NULL, 0);
```

The init function is called whenever an entry is created in the list to which the init function was assigned. This function call in this example causes the `myListEntryInit` function to be called so the entry can be initialized:

```
ADMEntryRef someEntry = sADMList->InsertEntry(myList, 0);
```

## Drawer functions

ADM dialogs, ADM items, and ADM entries can have custom draw handlers, which draw an object on the screen. Whenever an object needs to be updated, its ADM drawer is called. ADM drawers may enhance the

appearance of a standard object or perform all the drawing of an object. The draw function for an entry is set for its parent list and affects all the list's entries.

An ADM drawer function is defined as follows:

```
typedef void ADMAPI (*ADMObjectDrawProc)(ADMObjectRef inObject,
    ADMDrawerRef inDrawer);
```

The object reference is for the object to be drawn. The `ADMDrawerRef` is similar to a platform window reference or port and is where drawing commands are performed.

Drawing is done using the ADM Drawer suite, which contains a set of platform-independent graphics functions like `sADMDrawer->SetADMColor` and `sADMDrawer->DrawLine`. The `ADMDrawerRef` passed to the draw function is passed to each of the graphics functions.

This example of an ADM drawer calls the default draw function for the item, then supplements it by drawing a shadow rectangle around it:

```
void mySquareDrawHandler(ADMItemRef item, ADMDrawerRef drawer) {
    ADMRect boundsRect;
    sADMItem->DefaultDraw(item, drawer);

    sADMDrawer->GetBoundsRect(drawer, &boundsRect);
    boundsRect.top -= 2;
    boundsRect.bottom += 2;
    boundsRect.left -= 2;
    boundsRect.right += 2;

    sADMDrawer->SetADMColor(drawer, kADMShadowColor)
    sADMDrawer->DrawRect(drawer, &boundsRect)
}
```

This drawer example is assigned to an item in the code example in "Init functions" on page 42.

## Notifier functions

Notifiers probably are the event you will most often override. A notifier essentially is a notification that a high-level system event has occurred. Notifier events occur when a user interacts with an ADM object. Two common notifications are when a dialog is resized or an OK button is clicked. The latter often is how settings are extracted from an ADM modal dialog before the dialog is disposed of. ADM notifiers are listed and described in the table in "Notifier types" on page 98.

An ADM notifier function receives a reference to the object being notified and a notifier reference. The notifier reference is to the event that triggered it. The signature for the callback looks like this:

```
typedef void ADMAPI (*ADMObjectNotifyProc)(ADMObjectRef inObject,
    ADMNotifierRef inNotifier);
```

It would be used something like this:

```
void myOKHandler(ADMItemRef item, ADMNotifierRef notifier) {
    sADMItem->DefaultNotify(item, notifier);
    getDialogValues();
}
```

The `sADMItem->DefaultNotify` function call is made to provide the item's standard behavior. The dialog values would be extracted with other ADM Item suite functions. The above notify handler is assigned to a button in the code example in "Init functions" on page 42.

Dialog items often interact with each other; for instance, a button might restore the default values of other items. Here is an example of a notifier to accomplish this:

```
void mySetDefaultsButtonHandler(ADMItemRef item, ADMNotifierRef notifier) {
    ADMDialogRef thisDialog;
    ADMItemRef mySlider;

    sADMItem->DefaultNotify(item, notifier);

    thisDialog = sADMItem->GetDialog(item);
    mySlider = sADMDialog->GetItem(thisDialog, kMySliderItem);

    sADMItem->SetIntValue(mySlider, kDefaultSliderValue);
}
```

Notice that the handler function for the item does not need to use global references to the item with which it interacts. Instead, it gets its dialog, then uses this reference to obtain the other item.

The notifier reference passed to a notifier function can be used with the ADM Notifier suite to get more information about the reason for the notifier. For instance, several types of actions trigger a notify event:

```
#define kADMUserChangedNotifier             "ADM User Changed Notifier"
#define kADMBoundsChangedNotifier           "ADM Bounds Changed Notifier"
```

The changed notifier type is the most common reason a notifier is called; it simply means the user changed something in the dialog. The bounds changed notifier is received by an object when it is resized. To determine which event a dialog notifier has received, use the `sADMNotifier->IsNotifierType` function:

```
ASErr myResizeItemNotifyHandler(ADMItemRef item,
    ADMNotifierRef notifier)
{
    sADMItem->DefaultNotify(item, notifier);

    if (sADMNotifier->IsNotifierType(notifier, kADMBoundsChangedNotifier))
    {
        ADMDialogRef dialog = sADMItem->GetDialog(item);
        handleWindowResize(dialog);
    }
}
```

## Tracker functions

A tracker function is used by an ADM object to monitor low-level user events, like mouse movement and keystrokes, while it is the current object. In most cases, a notifier is sufficient, but when this is not enough, ADM dialogs, items, and entries can have trackers. List-entry trackers are set by the parent list and affect all its entries.

An ADM event tracking function is defined as follows:

```
typedef ADMBoolean ADMAPI (*ADMObjectTrackProc)(ADMObjectRef inObject,
    ADMTrackerRef inTracker);
```

The `ADMTrackerRef` basically is an identifier for the current event. The object reference is for the object receiving the event. If the track function returns `true`, its item receives a notify event when the mouse is released. For trackers on text items and key events, returning `true` means the key was handled. If it returns `false`, a notify event is not received.

Information about the event is obtained using the ADM Tracker suite functions. The `ADMTrackerRef` argument is passed to a function in the suite, and event information is returned.

This example of an ADM tracker function checks for and handles a shift click. A normal click is handled by the button's notifier function:

```
ADMBoolean mySquareTrackHandler(ADMItemRef item, ADMTrackerRef tracker)
{
    ADMBoolean shiftKeyDown, notify = true;
    ADMAction thisAction;

    shiftKeyDown = sADMTracker->GetModifiers(tracker) == kADMShiftKeyDownModifier;
    thisAction = sADMTracker->GetAction(tracker);

    if ((action == kADMButtonDownAction) && shiftKeyDown)
    {
        handleShiftClick();
        sADMTracker->Abort(tracker);
        notify = false;
    }
    return notify;
}
```

The tracker-function example above is assigned to an item in the code example in "Init functions" on page 42.

### Destroy functions

A destroy handler function is where you do any necessary clean up for an object about to be deleted from memory. It is triggered by a plug-in calling the `Destroy` function on the object. A destroy function is passed a reference to the object about to be destroyed and is defined as follows:

```
typedef void ADMAPI (*ADMObjectDestroyProc)(ADMObjectRef inObject);
```

If an init function allocated memory, it should be de-allocated here.

### Resizable windows

If a resizable window grows or shrinks, the resizing and relocating of dialog items must be handled. This event is sent to the dialog's resize-items notifier function, so adding your own notify handler and checking that the notify event type is an `kADMBoundsChangedNotifier` event allows you to handle the resize. For more information, see "Notifier functions" on page 43.

## Adding custom item types

**CAUTION:** Custom item types are deprecated.

A mechanism is provided through which new ADM item types can be added. These custom item types are then usable by other clients in their ADM dialogs. The provider is responsible for drawing the item for all subscribers and maintaining information needed to do so.

An example of a custom ADM Item can be seen in the Adobe Illustrator application tool panel. The fill and stroke color indicators at the bottom of the tool panel are a custom ADM Item provided by the paint style plug-in.

To add a new ADM item type, use `sADMDialog->RegisterItemType` at start-up:

```
error = sADMDialog->RegisterItemType(gPlugInRef, kMyADMCustomType);
```

When any client creates a dialog with the added custom type, the provider receives a PICA event to create it in an ADM dialog window. The provider is called through its main entry point with the following information:

```
caller == kADMCaller
selector == kADMCreateCustomItem
message == ADMCreateCustomItemMessage*
```

The `ADMCreateCustomItemMessage` data structure looks like this:

```
typedef struct
{
    SPMessageData d;
    struct ADMDialog *dialog;
    ADMint32 itemID;
    ADMItemType itemType;
    ADMRect boundsRect;
    ADMItemInitProc initProc;
    ADMUserData data;
    struct ADMItem *item;
}
ADMCreateCustomItemMessage;
```

The provider needs to respond to this event by creating an item within the indicated dialog, in the location specified by the bounds rectangle (*boundsRect*) in the message structure. This is done with the `sADMItem->Create` function:

```
message->item = sADMItem->Create(message->dialog, message->itemID,
    kADMUserType, &message->boundsRect, initProc, userData, options);
```

*All the event-handler functions for the new item must be overridden*. The new handlers would handle the drawing and user events for the item. In the above example, an ADM User type is used. You may be able to use another ADM Item type, if you want to use some of its functionality.

If a plug-in provides the custom item, it is responsible for acquiring itself to ensure it is not unloaded from memory. This is done using the PICA constant, `kSPAccessSuite`. When it no longer is supporting any ADM Items, the providing plug-in can release itself and be unloaded.

When the item is no longer needed (it or its containing dialog is to be destroyed), the destroy function that provided is called. At this point, you need to do any clean-up associated with the item.

## Using custom items

If a plug-in wants to use a custom item, it is a much simpler process. The plug-in simply includes an item of type `kADMCustomType` in its resource item list. When ADM loads and creates the dialog, it notices the custom item and tells the providing plug-in to create and maintain it. To indicate a custom item, the resources listed in the following tables are used, depending on the platform.

| Windows class name | Window style |
|---|---|
| *<custom item name>* | "" |

| Mac OS dialog item | CDEF res ID | Control title |
|---|---|---|
| Control Item | 1090 | <custom item name> |

The custom item also can be created at runtime with the `sADMDialog->CreateItem` function:

```
sADMDialog->CreateItem(myDialog, kMyCustomItemID, customItemType,
    &myCustomItemRect, itemInitProc, userData, options);
```

To initialize it, the custom item can use standard ADM Item suite functions, a function suite made available by the providing plug-in, or a combination of both.

# Using timer procedures

Often it is useful or necessary to use a timer function to provide a time-out or institute a custom reaction to the activities (or non-activities) of the plug-in user. ADM supports this need with several timer functions available in the suites. They all operate the same way.

An ADM timer function takes an item or dialog reference, a duration in milliseconds, and two callback procedures:

- The first callback procedure, the completion proc, is called if the duration expires. From the completion proc, you can return `true` and ADM repeats the timer.
- The second callback is called if the timer is aborted before the time duration elapses. An abort mask is passed to the timer create procedure and takes an action mask as defined in `ADMTracker.h`. If one of the actions in the mask occurs before the timer duration is finished, the abort proc is called. The action that caused the timer abort is passed to the callback.

# Using the C++ interfaces

There are two sets of interfaces for working with ADM, a standard C interface and a C++ interface. The C++ interface puts object-oriented wrappers around the procedural APIs, and you may find them more convenient to use. They eliminate the need to specify and de-reference the suite pointer and to pass the ADM object being processed. These wrappers can be found in the `IADM` (Interface to ADM) files in several Adobe SDKs.The following example shows the same process in both styles:

```
// Using ADM in a procedural manner
ADMItemRef theItem;
ADMListRef theList;

theItem = sADMDialog->GetItem(dialogRefkADMMenuItemID);
theList = sADMItem->GetList(theItem)
sADMList->SetMenuID(theList, gPlugInRef, 16000, "Choices");
sADMItem->Enable(theItem, true);

// Using ADM as objects
IADMItem theItem = IADMDialog::GetItem(kADMMenuItemID);
IADMList theList = theItem.GetList();
theList.SetMenuID(kColorMenuID);
theItem.Enable(true);
```

There are ADM C++ wrapper classes for each ADM object event suite. Each of these classes basically repackages a corresponding set of suite functions into an object definition; for instance:

```
class IADMDialog
class IADMItem
class IADMList
class IADMEntry

class IADMNotifier
class IADMDrawer
class IADMTracker
```

The convention used for the definition files is to add "I" (for "Interface") at the beginning of the standard ADM suite (for example, `IADMItem.hpp`).

In addition to the wrapper classes, base classes are provided to help create C++-based ADM dialog, two of which are as followss:

```
class BaseADMDialog
class BaseADMItem
```

These classes provide the basic constructors and destructors for ADM objects and a means for overriding event callbacks. There is no support for custom-event functions in the interface wrappers; these are handled in the base classes. You can use these as a foundation for building your own dialogs. The source and header files for these classes are provided in `.cpp` and `.hpp` files of the class name.

To use the C++ interfaces you must use the following globals for ADM suites:

```
ADMBasicSuite *sADMBasic;
ADMDialogSuite *sADMDialog;
ADMDialogGroupSuite *sADMDialogGroup;
ADMDrawerSuite *sADMDrawer;
ADMEntrySuite *sADMEntry;
ADMHierarchyListSuite *sADMHierarchyList;
ADMIconSuite *sADMIcon;
ADMImageSuite *sADMImage;
ADMItemSuite *sADMItem;
ADMListSuite *sADMList;
ADMListEntrySuite *sADMListEntry;
ADMNotifierSuite *sADMNotifier;
ADMTrackerSuite *sADMTracker;
```

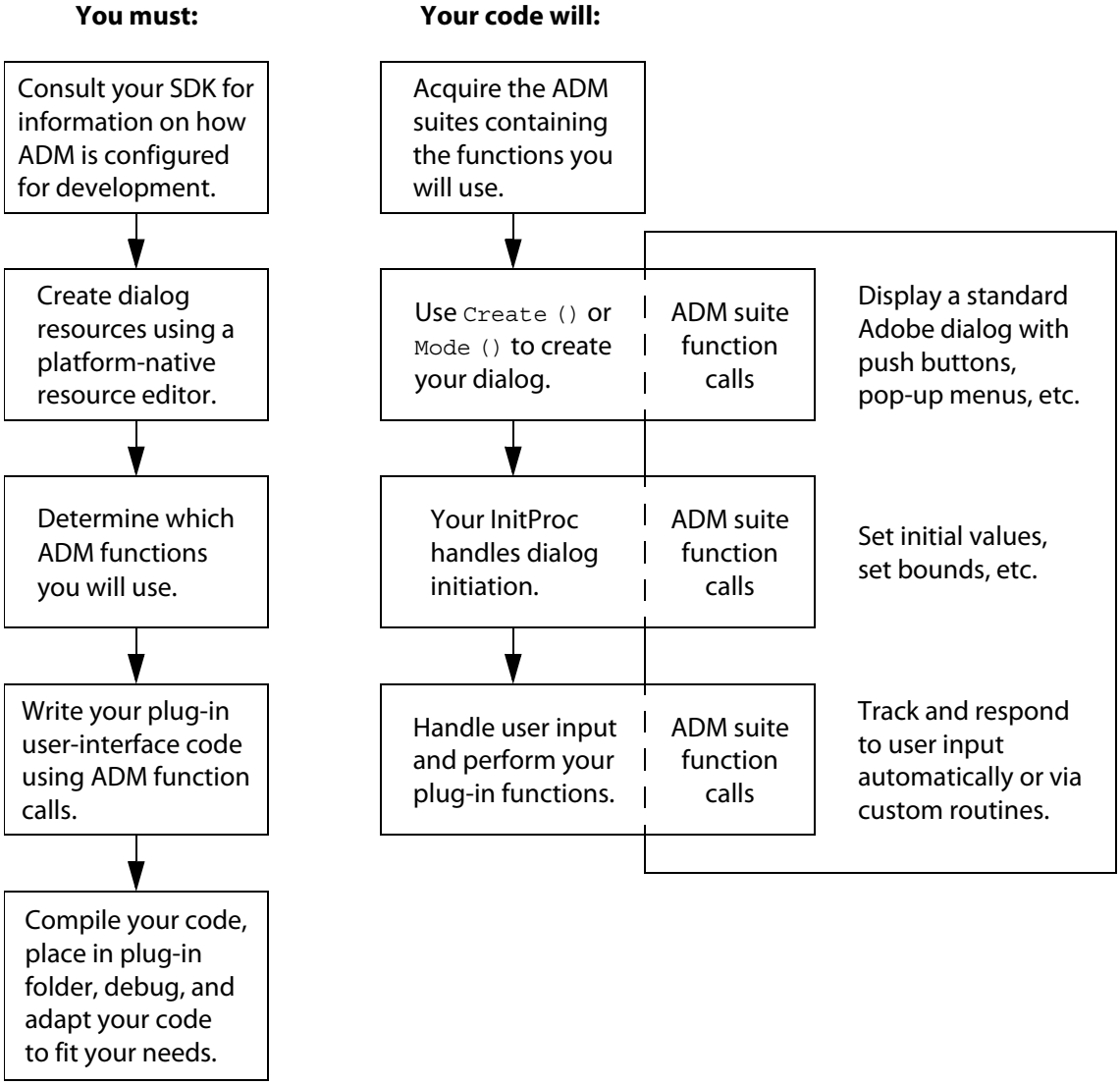
## Getting started with ADM plug-in development

The easiest way to get started quickly with ADM is to examine the sample code in the various SDKs available for the host applications using ADM (for example, Photoshop and Illustrator) and adapt it to your needs.

Once you have implemented the basic dialogs, you can refine and add any of the many features supported by ADM.

## General development process

The process of using ADM is illustrated below.




By using ADM, you can greatly reduce the time required to create robust dialogs that conform to the host-application appearance. This frees you to focus your energies on your plug-in functionality.

There are some basic requirements to using ADM. For example, for some applications the ADM plug-in module must be placed in the host application's plug-in folder to be available to the plug-in.

Second, whatever ADM features your plug-in uses must be acquired via PICA suite calls. Generally this should be the first thing your plug-in code does. Then, when your plug-in calls for an ADM feature, the appropriate ADM functions already are loaded into memory and ready to go. Typically, you acquire only the suites you actually need.

While ADM eliminates much of the work associated with handling dialogs and is supported on both Windows and Mac OS, the actual visual dialog resources themselves must be created using platform-specific resource editors like Microsoft® Visual Studio in Windows and Resorcerer and ResEdit in Mac OS. Once the resources are created, ADM handles displaying and manipulating them.

In general, your code does the following ADM-related tasks:

- Acquire the ADM suites that contain the functions you will use in your plug-in.
- Use `sADMDialog->Create` or `sADMDialog->Modal` to create your dialog
- Use an Init proc to set up your initial values and parameters
- Use `Tracker` and `Notify` functions to keep track of user events like mouse overs, mouse clicks, text entries, radio button selections, and popup-menu selections.
- Use `Destroy` to release your dialog (if it is a modeless or floating dialog).

You can make your dialogs as complex as you want and add custom graphics, custom event handlers, and so on, but ADM provides a rich set of automatic user-event handling, and you may find much of what you need already is built in to ADM.

# 2 Using ADM with Adobe Photoshop

Photoshop does not support all the functionality of ADM. In particular, it supports only modal dialogs and does not support modeless or floating/tabbed dialogs. ADM primarily is used in Photoshop to present dialog boxes for use with utility functions available from menu entries.

This chapter explains how the ADM plug-in code works for a specific Photoshop plug-in. To work through this chapter, use Photoshop 6.0 or later and the Photoshop 6.0 SDK or later (available from http://adobe.com/go/acrobat_developer). You can use these products on Windows or Mac OS.

## Frame Select Photoshop plug-in

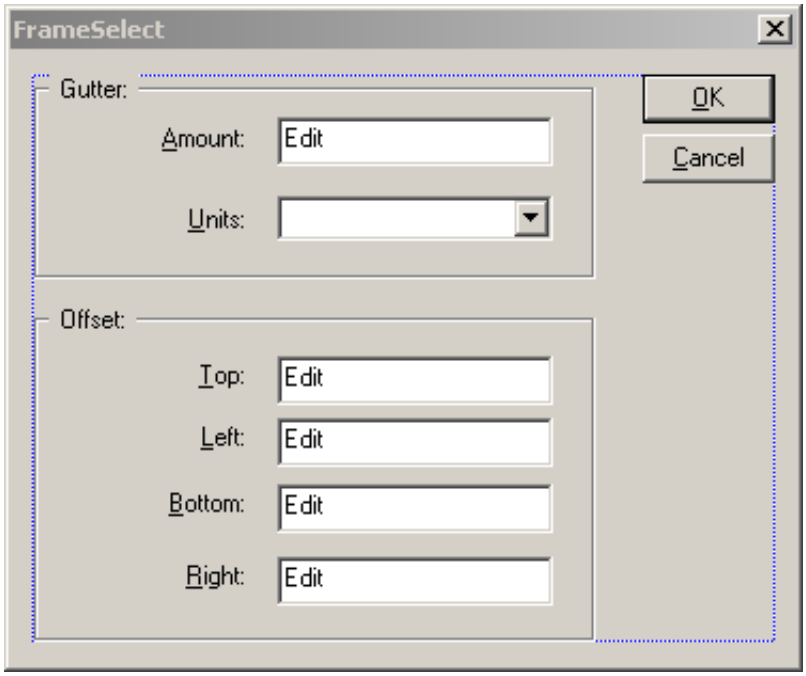
The example used in this chapter, Frame Select, is a sample automation plug-in available in the `SampleCode` directory of the Photoshop SDK. Automation plug-ins are used to execute Photoshop commands in a programmatic fashion. While designed as a simple sample of an automation plug-in that uses the scriptable actions engine in Photoshop, the Frame Select plug-in also is a good example of how to easily create and manage a Photoshop standard modal dialog using ADM.

## Platform-specific resources

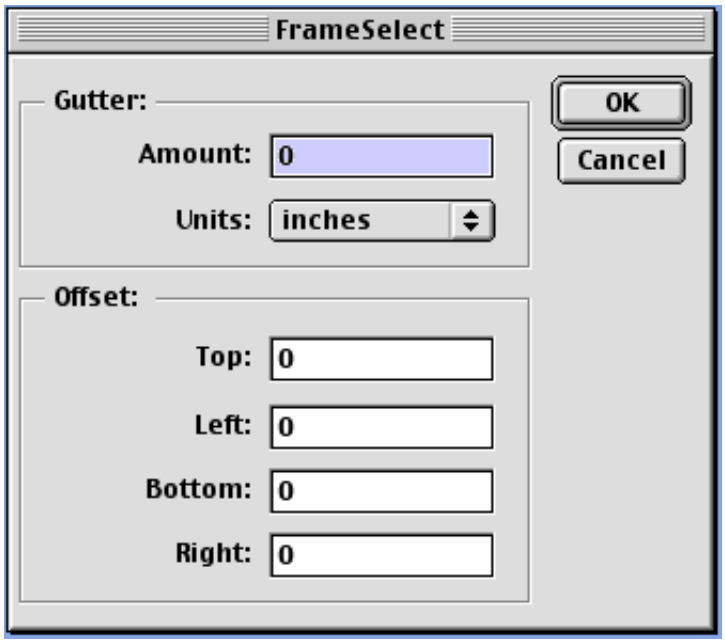
The first step in writing a plug-in that uses ADM is to create the visual dialog itself, using a resource editor. On Windows, tools available in the Visual Studio development environment commonly are used. On Mac OS, a product like Resorcerer from Mathemaesthetics, Inc. can be used.

The code generated in this process is the only platform-specific code you need to write an ADM-based plug-in that executes on both Windows and Mac OS. Once the code is written, ADM executes it using the `dialogID` argument to the `sADMDialog->Modal` function; this is the platform native resource ID that is used in creating the dialog.

For the Frame Select plug-in, the dialog shown below was created on Windows.




And the dialog shown below was created on Mac OS.




The Frame Select plug-in pops these dialogs, filling the various options with default values (shown). It then accepts any user changes or inputs. On dismissal of the dialog via the OK button, it creates a new frame in the Photoshop document. This frame can be used to edit the document as required.

The dialogs contain static-text items, text-edit items, and the OK and Cancel push buttons. The following is the Windows platform-specific code used to generate the Windows dialog shown above. It can be found in the FrameSelect.rc file in the Frame Select Visual Studio project:

```
16001 DIALOG DISCARDABLE  0, 0, 256, 191
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "FrameSelect"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON       "&OK",1,208,7,41,14
    PUSHBUTTON          "&Cancel",2,208,24,41,14
    RTEXT               "&Amount:",5,17,21,56,11
    EDITTEXT            6,83,19,86,14,ES_AUTOHSCROLL
    RTEXT               "&Units:",7,17,44,56,11
    COMBOBOX            8,83,42,86,17,CBS_DROPDOWNLIST | WS_TABSTOP
    GROUPBOX            " Gutter: ",3,7,7,175,59
    RTEXT               "&Top:",11,17,89,56,11
    EDITTEXT            12,83,88,86,14,ES_AUTOHSCROLL
    RTEXT               "&Left:",13,17,107,56,11
    EDITTEXT            14,83,106,86,14,ES_AUTOHSCROLL
    RTEXT               "&Bottom:",15,17,126,56,11
    EDITTEXT            16,83,125,86,14,ES_AUTOHSCROLL
    RTEXT               "&Right:",17,17,147,56,11
    EDITTEXT            18,83,146,86,14,ES_AUTOHSCROLL
    GROUPBOX            " Offset: ",9,7,87,175,97
END

IDD_DIALOG1 DIALOG DISCARDABLE  0, 0, 186, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Dialog"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "OK",IDOK,129,7,50,14
    PUSHBUTTON     "Cancel",IDCANCEL,129,24,50,14
END

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    16001, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 249
        TOPMARGIN, 7
        BOTTOMMARGIN, 184
END
```

In Mac OS, the `FrameSelectMacResources.r` file is the resource text file in the Frame Select CodeWarrior project that describes the Mac OS dialog shown above. It contains the Mac OS `DLOG` and `DITL` dialog-box information. The `DLOG` resource describes all dialog-box information (size, visibility, whether it has a "go away" box, etc.), while the `DITL` resource describes the individual elements within the dialog box (each button, text field, etc.). The following code generates the Mac OS dialog shown above.

```
resource 'DLOG' (kDialogID, plugInName " UI", purgeable) {
    {190, 203, 450, 523},
    movableDBoxProc,
    visible,
    noGoAway,
    0x0,
    kDialogID,
    plugInName,
    centerParentWindowScreen
};
```

```
resource 'DITL' (kDialogID, plugInName " UI", purgeable) {
    {   /* array DITLarray: 18 elements */
        /* [1] */
        {8, 252, 28, 312},
        Button {
                  enabled,
                  "OK"
        },
        /* [2] */
        {34, 252, 56, 312},
        Button {
                  enabled,
                  "Cancel"
        },
        /* [3] */
        {9, 12, 29, 64},
        StaticText {
                  enabled,
                  " Gutter:"
        },
        /* [4] */
        {17, 0, 96, 240},
        UserItem {
                  enabled
        },
        /* [5] */
        {34, 36, 54, 108},
        StaticText {
                  disabled,
                  "Amount:"
        },
        /* [6] */
        {35, 120, 51, 220},
        EditText {
                  enabled,
                  ""
        },
        /* [7] */
        {65, 36, 85, 108},
        StaticText {
                  disabled,
                  "Units:"
        },
        /* [8] */
        {63, 117, 83, 223},
        Control {
                  enabled,
                  16001
        },
        /* [9] */
        {101, 12, 121, 64},
        StaticText {
                  enabled,
                  " Offset:"
        },
        /* [10] */
        {109, 0, 253, 240},
        UserItem {
                  enabled
        },
```

```
        /* [11] */
        {129, 36, 149, 108},
        StaticText {
                  disabled,
                  "Top:"
        },
        /* [12] */
        {130, 120, 146, 220},
        EditText {
                  enabled,
                  ""
        },
        /* [13] */
        {161, 36, 181, 108},
        StaticText {
                  disabled,
                  "Left:"
        },
        /* [14] */
        {162, 120, 178, 220},
        EditText {
                  enabled,
                  ""
        },
        /* [15] */
        {191, 36, 211, 108},
        StaticText {
                  disabled,
                  "Bottom:"
        },
        /* [16] */
        {192, 120, 208, 220},
        EditText {
                  enabled,
                  ""
        },

        /* [17] */
        {221, 36, 241, 108},
        StaticText {
                  disabled,
                  "Right:"
        },
        /* [18] */
        {222, 120, 238, 220},
        EditText {
                  enabled,
                  ""
        }
    }
};
```

```
resource 'CNTL' (kDialogID, "Units", purgeable) {
    {16, 4, 36, 110},
    0,
    visible,
    0,
    kDialogID,
    1016,
    0,
    ""
};

resource 'STR#' (kResetStringID, purgeable)
    {
        {
            "Reset"
        }
    };

resource 'STR#' (kCancelStringID, purgeable)
    {
        {
            "Cancel"
        }
    };

resource 'STR#' (kUnitsID, purgeable)
{
    {
        "pixels",
        "inches",
        "cm",
        "points",
        "picas",
        "percent"
    }
};
```

# Acquiring the suites

Before the plug-in code can present the dialog to the user, it must acquire the ADM suites. The ADM suites become available when the ADM plug-in is loaded as Photoshop launches; the process of acquiring the suites is a process of acquiring pointers that can be used to invoke the functions. This is accomplished using the `PIUSuitesAcquire` function, declared in the `PIUSuites.h` file, which is found in the SampleCode/Common directory in the Photoshop SDK. The function has the following signature:

```
SPErr PIUSuitesAcquire(SPBasicSuite* sSPBasic, _AcquireList*
suitesToAcquire, int16 numSuites);
```

The `PIUSuitesAcquire` function takes three arguments, described in the following table.

| Argument | Description |
|---|---|
| sSPBasic | A pointer to the SPBasicSuite struct. This struct contains two host member functions, AcquireSuite and ReleaseSuite, which are invoked by PIUSuitesAcquire to acquire the suites. |
| suitesToAcquire | A pointer to the _AcquireList struct, which is used to pass in the list of suites that need to be acquired and to get the pointers to the suites. Like PIUSuitesAcquire, it is defined in PIUSuites.h:<br>`typedef struct`<br>`{`<br>`  char* name;`<br>`  int16 version;`<br>`  void** suitePtr;`<br>`} _AcquireList;` |
| numSuites | The number of suites to be acquired. |

In the Frame Select plug-in code, the call to this function is made in the FrameSelect.cpp file. An instance of the _AcquireList structure is created (called MySuites) and populated with the names and versions of the suites to be acquired, and with void placeholder pointers to the suites that will be filled by the call to PIUSuitesAcquire.

The sSPBasic pointer is passed to the plug-in when it loads, along with the gPlugInRef, which is the unique plug-in ID.

# Building, presenting, and using the dialog

The FrameSelectUI.cpp file contains the code used to initialize and present the dialog and to set up the notifications and callbacks for the dialog.

FrameSelectUI.cpp includes an enumerated list of the constants of all dialog items, starting with the OK button as item 1 and the Cancel button as item 2.

**NOTE:** Unless otherwise specifically changed by the programmer using a "make default item" function, ADM expects that the default item is the first item and acts as the OK button, and item 2 is the Cancel button.

Below is the enumerated list of dialog items. After assigning a value to the first item, subsequent items automatically are assigned the next number in order.

```
enum
    {
    kDNoUI = -1, // Error.
    kDOk_button = 1, // Must be one.
    kDCancel_button, // Must be two.
    kDGutter_staticText,
    kDGutter_frame,
    kDAmount_staticText,
    kDAmount_editText,
    kDUnits_staticText,
    kDUnits_popUp,
    kDOffset_staticText,
    kDOffset_frame,
    kDTop_staticText,
    kDTop_editText,
    kDLeft_staticText,
    kDLeft_editText,
    kDBottom_staticText,
    kDBottom_editText,
    kDRight_staticText,
    kDRight_editText
    };
```

The `k` designation indicates a constant. The `D` designation is an Adobe convention used to indicate that these constants are dialog-box related.

**NOTE:** The order follows the numbering of the items in the `resource 'DITL'` on Mac OS. There is a similar correspondence on Windows.

The dialog is popped in the `DoUI` function. For a description of the parameters, see the `sADMDialog->Modal` function. One of the parameters that is passed is the initialization function. For this plug-in, the name of the initialization function is `DoUIInit`, and it follows the calling conventions for the `ADMDialogInitProc` callback, as documented in `sADMDialog->Modal`. The initialization function initializes each item in the dialog, provides focus to the first text field (Amount:) in the dialog, and installs a tracker to change other items as the user enters text into the first text field. Since the code required to initialize the Cancel button and the pop-up menu is extensive, separate functions are used for these tasks.

The `DoUI` function handles interaction with the user-interface items that make up the dialog. If the user presses the Cancel button, `DoUI` restores the original frame parameter values and exits. Otherwise, the parameters are updated based on any changes the user made while interacting with the dialog. See the following code:

```
SPErr DoUI ()
    {
    SPErr error = noErr;
    int item = kDNoUI; // Error value.
    SaveParameters(); // Save our parameters, just in case.

    if (sADMDialog != NULL)
        {
        item = sADMDialog->Modal
            (
                gPlugInRef,
                "FrameSelection",
                kDialogID,
                kADMModalDialogStyle,
                DoUIInit,
                NULL, /* No user data */
                0//InOptions = 0
            );
        }

    if (item != kDOk_button)
        {
        error = 'STOP';
        RestoreParameters();
        }

    return error;

    }
```

If the `sADMDialog` pointer was set to point to the suite, the modal dialog is generated using the `sADMDialog->Modal` function call. If the call is successful, `item` is assigned a valid value by the `sADMDialog->Modal` function call. This value is the number of the dialog item used to dismiss the dialog, usually the `kDOk_button` or `kDCancel_button` (items 1 and 2). If it is not the `kDOk_button`, the original frame parameters are restored, and the appropriate error is returned.

The `sADMDialog->Modal` function, as coded here, takes the parameters listed in the following table.

| Parameter | Value | Description |
|---|---|---|
| `pluginRef` | `gPlugInRef` | Several ADM suites and functions require the plug-in reference of the current plug-in. ADM uses that information to track who owns what dialog. The value assigned to the global `gPlugInRef` is passed in by the host application via `PluginMain`, which is called when the plug-in is first loaded. |
| `name` | `FrameSelection` | The name of the new dialog. In this case, we choose FrameSelection. |
| `dialogID` | `kDialogID` | A dialog ID (`uiID`) is provided. It is the same ID as used in the `FrameSelect.r` file. This tells ADM what resource to use for the dialog. The cross-platform ADM routines draw the user interface using the platform-specific dialog resources discussed earlier in this chapter. In our plug-in, `uiID` is defined as `16000`. (In Mac OS, see `PIUtilities.r` for those define statements.) |

| Parameter | Value | Description |
|---|---|---|
| style | kADMModalDialogStyle | A constant that specifies the style of dialog. In this case (and most Photoshop plug-ins), kADMModalDialogStyle is used. This dialog style supports a window that can be moved around the screen and must be dismissed before continuing. |
| initProc | DoUIInit | The name of the ADMDialogInitProc callback function that initializes the dialog elements using ADM routines. In this case, it is the DoUIInit function. |
| data | NULL | Any extra data that may be required. This information can be a pointer, strings, structures, or whatever you may want to associate with your dialog boxes. In this case, there is no user data, so we pass NULL. |
| inOptions | 0 | Not used here. See sADMDialog->Modal. |

The first function executed when DoUI is run is DoUIInit:

```
static ASErr ASAPI DoUIInit(ADMDialogRef dialog)
{
    // SPErr = Sweet Pea / PICA error type;
    // OSErr = Photoshop uses the Operating system type;
    // ASErr = SPErr = "Adobe Systems" error type;
    ASErr error = kSPNoError;

    ADMItemRef item;

    // Set up list and display default item:
    InitCancelButton(dialog);

    // Set up pop-up menu:
    InitUnitsPopUp(dialog);

    // Set up group menu to notify us when changed:
    item = sADMDialog->GetItem(dialog, kDUnits_popUp);
    sADMItem->SetNotifyProc(item, DoUnitsPopUp);

    // Set up Amount edit text:
    item = sADMDialog->GetItem(dialog, kDAmount_editText);
    sADMItem->SetMaxTextLength(item, kMaxTextLength);
    sADMItem->SetUnits(item, kADMNoUnits);
    sADMItem->SetPrecision(item, kPrecision);
    sADMItem->SetMinFloatValue(item, kTopMin);
    sADMItem->SetMaxFloatValue(item, kTopMax);
    sADMItem->SetNotifyProc(item, DoAmountEditText);

    // Install tracker to change other items as text is
    // edited:
    sADMItem->SetTrackProc(item, TrackAmountEditText);

    // Hilight edit text as first item:
    sADMItem->Activate(item, true);
    sADMItem->SelectAll(item);
```

```
    // Set up Top edit text:
    item = sADMDialog->GetItem(dialog, kDTop_editText);
    sADMItem->SetMaxTextLength(item, kMaxTextLength);
    sADMItem->SetUnits(item, kADMNoUnits);
    sADMItem->SetPrecision(item, kPrecision);
    sADMItem->SetMinFloatValue(item, kTopMin);
    sADMItem->SetMaxFloatValue(item, kTopMax);
    sADMItem->SetNotifyProc(item, DoTopEditText);

    // Set up Left edit text:
    item = sADMDialog->GetItem(dialog, kDLeft_editText);
    sADMItem->SetMaxTextLength(item, kMaxTextLength);
    sADMItem->SetUnits(item, kADMNoUnits);
    sADMItem->SetPrecision(item, kPrecision);
    sADMItem->SetMinFloatValue(item, kLeftMin);
    sADMItem->SetMaxFloatValue(item, kLeftMax);
    sADMItem->SetNotifyProc(item, DoLeftEditText);

    // Set up Bottom edit text:
    item = sADMDialog->GetItem(dialog, kDBottom_editText);
    sADMItem->SetMaxTextLength(item, kMaxTextLength);
    sADMItem->SetUnits(item, kADMNoUnits);
    sADMItem->SetPrecision(item, kPrecision);
    sADMItem->SetMinFloatValue(item, kBottomMin);
    sADMItem->SetMaxFloatValue(item, kBottomMax);
    sADMItem->SetNotifyProc(item, DoBottomEditText);

    // Set up Right edit text:
    item = sADMDialog->GetItem(dialog, kDRight_editText);
    sADMItem->SetMaxTextLength(item, kMaxTextLength);
    sADMItem->SetUnits(item, kADMNoUnits);
    sADMItem->SetPrecision(item, kPrecision);
    sADMItem->SetMinFloatValue(item, kRightMin);
    sADMItem->SetMaxFloatValue(item, kRightMax);
    sADMItem->SetNotifyProc(item, DoRightEditText);

    UpdateRectValues(dialog);

    // Set up justification and style for static text items:
    item = sADMDialog->GetItem(dialog, kDAmount_staticText);
    sADMItem->SetJustify(item, kADMRightJustify);

    item = sADMDialog->GetItem(dialog, kDUnits_staticText);
    sADMItem->SetJustify(item, kADMRightJustify);

    item = sADMDialog->GetItem(dialog, kDTop_staticText);
    sADMItem->SetJustify(item, kADMRightJustify);

    item = sADMDialog->GetItem(dialog, kDLeft_staticText);
    sADMItem->SetJustify(item, kADMRightJustify);

    item = sADMDialog->GetItem(dialog, kDBottom_staticText);
    sADMItem->SetJustify(item, kADMRightJustify);

    item = sADMDialog->GetItem(dialog, kDRight_staticText);
    sADMItem->SetJustify(item, kADMRightJustify);

    return error;
}
```

This function starts by initializing its error condition as no error. Then, a variable item of type `ADMItemRef` is declared. ADM expects all dialog objects to have their own native type, so it can perform the appropriate type checking. In this case, we create a reference to a type `ADMItem`. Other possible ADM types include `ADMDialog`, `ADMDrawer`, `ADMList`, `ADMEntry`, `ADMNotifier`, `ADMTracker`, `ADMIcon`, `ADMImage`, `ADMUserData`, `ADMTimer`, `ADMActionMask`, and `ADMChar`. These are defined elsewhere in this document and listed in the `ADMTypes.h` file.

Next, this function sets up the Cancel button to notify our user-interface routine when it is clicked by the user. Since we want to be able to support the Photoshop convention of using the Cancel button as a "reset to default parameters" function (when the user holds down the Option/Alt key when pressing Cancel), we must be notified when the Cancel button is pressed. The call to `InitCancelButton(dialog)` sets up these parameters. `InitCancelButton` is discussed after `DoUIInit`.

Next, the group pop-up menu is created. First the list is cleared, then a new list is created. A notification is set for when the user changes units.

After that, the text-edit dialog boxes are created for Amount:, Top:, Left:, Bottom:, and Right:. Focus is provided to the first text field (Amount:), and a tracker is installed to simultaneously update the other text fields as the user enters text into the Amount: text field.

The various options (maximum text length, units, precision, minimum value, and maximum value) are handled using ADM function calls, as shown below for the Amount: dialog. This is accomplished using the `sADMItem->SetMaxTextLength`, `sADMItem->SetUnits`, `sADMItem->SetPrecision`, `sADMItem->SetMinFloatValue`, and `sADMItem->SetMaxFloatValue` functions.

```
// Set up Amount edit text:
item = sADMDialog->GetItem(dialog, kDAmount_editText);
sADMItem->SetMaxTextLength(item, kMaxTextLength);
sADMItem->SetUnits(item, kADMNoUnits);
sADMItem->SetPrecision(item, kPrecision);
sADMItem->SetMinFloatValue(item, kTopMin);
sADMItem->SetMaxFloatValue(item, kTopMax);
sADMItem->SetNotifyProc(item, DoAmountEditText);
```

Finally `sADMItem->SetNotifyProc` is used to set the callback that is invoked when the user types text into the text box. The callback in this example is named `DoAmountEditText`, and it follows the format defined for callbacks of type `ADMItemNotifyProc`. The format for `ADMItemNotifyProc` is defined in the documentation for `sADMItem->SetNotifyProc`.

Notifiers are important ADM functions that are explained further in later chapters. ADM reports on several different user actions and provides this information via notify routines. ADM Notifiers track several user events: User Changed, Entry Text Changed, Close Hit Notifier, Collapse Notifier, Expand Notifier, Bound Changed, Hide Window Modifier, etc. `ADMNotifier.h` defines all available ADM notifiers. The User Changed Notifier is most helpful, since it notifies our routines that the user changed something in our dialog boxes.

The `InitCancelButton` function initializes the cancel button to receive notification when the user presses the Cancel button. In this plug-in, we want to support the Photoshop convention of allowing dual use of the Cancel button. When the user presses the Option key while clicking Cancel, the function performed is "reset to default values," and the Cancel button displays the Reset label instead of the Cancel label. The dialog routines must be notified when this condition occurs. ADM makes it possible to track this user activity and change the button labels on the fly.

`InitCancelButton`, shown below, takes a reference to the current dialog box and returns any error that occurs while initializing the Cancel button. The routine assigns a notifier proc for the Cancel button to trap the Option/Alt key and mouse-down notifiers and call routines when these events happen.

```
static SPErr InitCancelButton(ADMDialogRef dialog)
    {
    SPErr error = kSPNoError;

    if (dialog != NULL)
        {
        // Set up "Cancel" button to notify us when its been clicked:
        ADMItemRef item = sADMDialog->GetItem(dialog,
            kDCancel_button);
        if (item != NULL)
            {
            sADMItem->SetNotifyProc(item, DoCancel);

            // Set up name of Cancel button. Since we have to have
            // resources around for "Reset" and "Cancel", we might
            // as well check them to load the right value:
            if (gCancelButtonIsReset)
                SetTextToReset(item);
            else
                SetTextToCancel(item);

            // Set up mask for tracker function:
            ADMActionMask mask = sADMItem->GetMask(item);
            sADMItem->SetMask
                (
                item,
                mask |
                kADMModKeyDownMask |
                kADMModKeyUpMask |
                kADMLeaveMask |
                kADMEnterMask |
                kADMButtonUpMask
                );

            // Install tracker for it to be "Reset" when it needs to:
            sADMItem->SetTrackProc(item, DoReset);
            }
        else
            {
            error = kSPBadParameterError;
            }
        }
    else
        {
        error = kSPBadParameterError;
        }

    return error;

    }
```

The routine starts with a no-error assignment. It then checks to ensure the dialog is not `NULL`; then it uses the ADM Dialog suite to obtain a reference to the Cancel button of the current dialog:

```
ADMItemRef item = sADMDialog->GetItem(dialog, kDCancel_button);
```

We then set up the notifier to call our routine when the button is pressed:

```
sADMItem->SetNotifyProc(item, DoCancel);
```

Since we can have two conditions for our Cancel button (Cancel and Reset), we set the text to be displayed on the button using the `SetTextToCancel` or `SetTextToReset` functions depending on which condition is true, as specified by the state of the boolean variable `gCancelButtonIsReset`.

The `SetTextToCancel` function, shown below, gets the appropriate Cancel text and uses the `sADMItem->SetText` function to set the text label for the cancel button on the fly:

```
static void SetTextToCancel
    (
    /* IN */  ADMItemRef item // Item ID for Cancel button.
    )
    {
    if (item != NULL)
        {
        char text[256];
        sADMBasic->GetIndexString(gPlugInRef, kCancelStringID, 1, text, 256);
        sADMItem->SetText(item, text);
        } // item null
    }
```

Likewise, the `SetTextToReset` function, shown below, does the same thing for the `Reset` label:

```
static void SetTextToReset
    (
    /* IN */  ADMItemRef item // Item ID for Cancel button.
    )
    {
    if (item != NULL)
        {
        char text[256];
        sADMBasic->GetIndexString(gPlugInRef, kResetStringID, 1, text, 256);
        sADMItem->SetText(item, text);
        } // item null
    }
```

Finally, we must set the mask to be able to check whether the Option/Alt key was held down when the Cancel button was pressed. This mask information is set up using `sADMItem->SetMask`.

You set up a mask to tell ADM what events report to your tracker. This is done by declaring a mask, getting the current mask, and setting a new mask to track the additional events you want:

```
ADMActionMask mask = sADMItem->GetMask(item);
    sADMItem->SetMask
    (
    item,
    mask |
    kADMModKeyDownMask |
    kADMModKeyUpMask |
    kADMLeaveMask |
    kADMEnterMask |
    kADMButtonUpMask
    );
```

To track the Option/Alt key event, we use a tracker callback function. ADM must be set up to call our track routine when an event occurs:

```
sADMItem->SetTrackProc(item, DoReset);
```

We are choosing to track whether the Mod key (the Option key on Mac OS, and the Alt key on Windows) is up or down, whether the mouse leaves or enters the Cancel button, and when the user clicks. This lets us

control how the Cancel button turns into the Reset button (when the Option key is pressed and the mouse is over the Cancel button) and when the reset is performed (upon click with the Option/Alt key pressed).

This routine is called when the cancel button is pressed. It is a notifier-receiver:

```
static void ASAPI DoCancel
    (
    ADMItemRefitem,
    ADMNotifierRefnotifier
    )
    {
    if (sADMNotify->IsNotifierType(notifier, kADMUserChangedNotifier))
        { // Correct notifier.  Do this:
        if (gCancelButtonIsReset)
            { // Must be reset!
            RestoreParameters(); // Resets.
            ADMDialogRef dialog = sADMItem->GetDialog(item);
            DoUIInit(dialog);
            }
        else
            {
            sADMItem->DefaultNotify(item, notifier);
            }
        }
    else
        {
        sADMItem->DefaultNotify(item, notifier);
        }
    }
```

The statement `if (sADMNotify->IsNotifierType(notifier, kADMUserChangedNotifier))` determines what type of user input was performed. This is possible because ADM tracks several user actions and provides this information. `ADMNotifier.h` defines all available ADM notifiers. In our case, `DoCancel` obtains the type of notifier and tests it to see if the Cancel button is Reset (using the `gCancelButtonIsReset` boolean variable defined at the top of the `FrameSelectUI.cpp` file). If it is, the previous parameters are restored, and `DoUIInit` is called to stuff the default parameters into their text-edit fields and reset check and radio-button groups.

The `Do***EditText` callbacks are called when the text is changed, so the value can be acquired and put in a global variable to change the frame. The callbacks were set in the `DoUIInit` initialization function.

To acquire the new values in the dialog boxes, we use the ADM functions to get the value of the changed dialog box. For example, in the `ConvertEditTextNumber` function:

```
double value = (double)sADMItem->GetFloatValue(item);
```

This is a good example of how ADM simplifies the handling of our dialog boxes. In native code for Mac OS (or Windows) to obtain the value of the user input, we would have to acquire the string data, then convert to integer values, etc.

Finally, we use `SaveParameters` and `RestoreParameters` to save and restore our parameters.

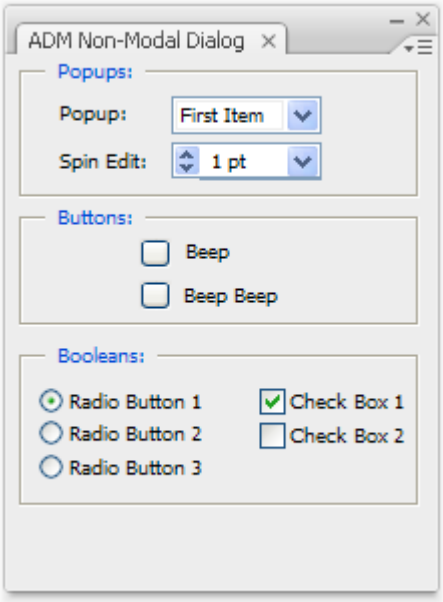
# 3 Using ADM with Adobe Illustrator

This chapter explains how the ADM plug-in code works for a specific Illustrator plug-in. To work through this chapter, you may want to have the Illustrator SDK on hand, so you can browse the code in the plug-in.

## ADMNonModalDialog plug-in

The Illustrator SDK provides a sample called ADMNonModalDialog that demonstrates the process of creating a modeless (or floating) dialog. While it does not do anything but put up a floating dialog, it shows all ADM function calls required to manage pop-up menus and lists. Unlike Photoshop, Illustrator supports all ADM functionality, including modal and modeless dialogs.

The following figure shows the dialog created by the ADMNonModalDialog sample.




This dialog is the floating, tabbed, panel type. If you have multiple tabbed dialogs, they can be docked together into a single dock. This behavior is handled by ADM and requires little support from your ADM dialog.

## Platform-specific resources

The first step is to create the visual dialog resource. You may want to start with the ADM Non-Modal Dialog plug-in resource and adjust its parameters to fit your needs.

In Mac OS, use a visual editor like Resorcerer from Mathemaesthetics, Inc to create your resources. In Windows, use Microsoft Visual Studio.

The resources that define this dialog's controls are in the sample's `plugin.rc` file (Windows) or `adm.rsrc` (Mac OS).

# Dialog creation

The dialog is created using the `sADMDialog->Create` function, as shown in the code below:

```
// Create the non-modal dialog. This does not necessarily show the dialog on
// the screen. If the dialog was hidden at last shutdown, it will not be
// shown until sADMDialog->Show() is called.
// Note: the Init proc, DlgInit, will be called immediately following
//     sADMDialog->Create()

AIErr createADMDialog(AINotifierMessage *message) {
       AIErr error = kNoErr;

       g->nonModalDialog = sADMDialog->Create(message->d.self,
              "ADMNonModalDialog", kADMNonModalDialogID,
              kADMTabbedFloatingDialogStyle, DlgInit, nil, 0);

       if (error)
              goto error;
error:
       return error;
}
```

# Dialog initialization

All ADM suites are included in the project in the `common.h` file, as follows:

```
// ADM Suites
#include "ADMBasic.h"
#include "ADMDialog.h"
#include "ADMItem.h"
#include "ADMIcon.h"
#include "ADMList.h"
#include "ADMDialogGroup.h"
#include "ADMNotifier.h"
#include "ADMEntry.h"
#include "ADMTracker.h"
```

In the ADM non-modal dialog, there are various pop-up menus, push buttons, radio buttons, and check boxes. The constants for these items are defined in the `admHandler.h` file that is included in the `admHandler.cpp` code. These constants are as follows:

```
#define kADMNonModalDialogID 16000

#define kpopupItem 5
#define kpopupMenuID 800

#define kbeepItem 3
#define kbeepBeepItem 4

#define kspinEditPopupItem 6
#define kspinEditPopuMenuID 900

#define kradioButton1Item 7
#define kradioButton2Item 8
#define kradioButton3Item 9

#define kcheckBox1Item 10
#define kcheckBox2Item 11
```

The `admHandler.h` file contains prototypes for all functions covered in the following discussion.

In `admHandler.cpp`, the dialog initialization procedure, `DlgInit`, starts with the declaration of various ADM variables, `ADMItemRef`, `ADMListRef`, `ADMEntryRef`, and so on:

```
ASErr ASAPI DlgInit(ADMDialogRef dlg)
{
    PUSH_GLOBALS
    ASErr                       fxErr = kNoErr;
    AIAppContextHandle          AppContext;
    ADMItemRef                  menuItemRef, popupItemRef, beepButtonItemRef,
                                beepBeepButtonItemRef, spinEditPopupItemRef;
    ADMListRef                  menuListRef, popupListRef, spinEditPopupListRef;
    ADMEntryRef                 entry;
    SPPluginRef                 pluginRef;
    char                        tipString[64];
    char                        groupName[64] = "SDK Dialogs";
    ADMPoint                    location, size;
    ADMRect                     rect, dimensions, boundsRect;
    long                        positionCode;
    tipString[0] = 0;

    // Set up the application context, so that suite calls will work
    pluginRef = sADMDialog->GetPluginRef(dlg);
    sAIAppContext->PushAppContext(pluginRef, &AppContext);

    // Acquire yourself to stay in memory
    sSPAccess->AcquirePlugin( pluginRef, &g->accessRef );

    // Acquire suites to stay in memory
    acquireSuites( g->basic );
```

Key items in this dialog are as follows:

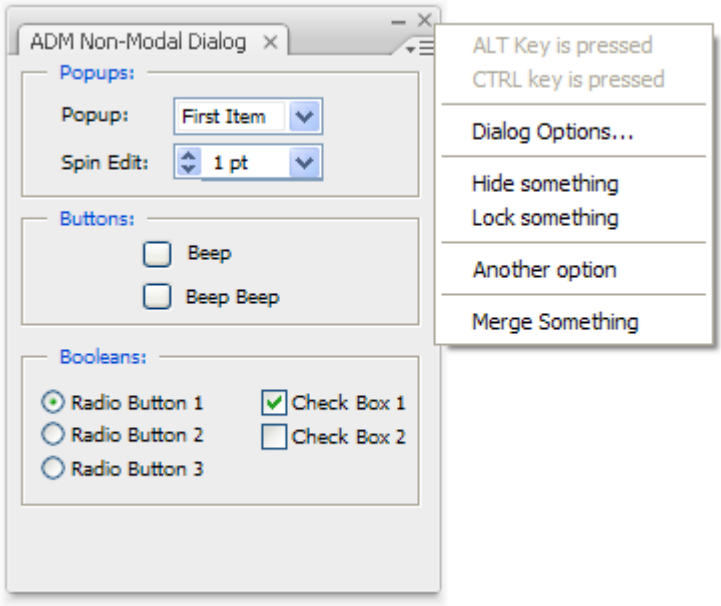
- A `kADMPopupMenuType` menu (the right arrow on the right side of the panel).
- A `kADMPopupListType` menu item (labeled Popup**:**).
- A `kADMSpinEditPopupType` menu item (labeled Spin Edit).
- Two `kADMTextPushButtonType` items (labeled Beep and Beep Beep).

In addition, the necessary list references are declared in the following line:

```
ADMListRef menuListRef, popupListRef, spinEditPopupListRef;
```

## Pop-up menu item

Note the arrow button at the top of the panel. This indicates a `kADMPopupMenuType` menu, as shown in the following figure.




The first two items are grayed out. They show the ability of ADM to support Mod key effects. In this case (Windows), they become active only when the Alt key is held down while the right-arrow menu button is clicked, or the Ctrl key is pressed while the menu arrow is clicked. (On Mac OS, the keys are Option and Command, respectively.)

The following code in `DlgInit` sets up this `kADMPopupMenuType` dialog item:

```
// Set up popup menu on dialog
menuItemRef = sADMDialog->GetItem(dlg, kADMMenuItemID);
sADMItem->SetNotifyProc(menuItemRef, dialogPopupMenuProc);
if (menuItemRef)
{
    menuListRef = sADMItem->GetList(menuItemRef);
    if (menuListRef)
    {
        // 700 is the MENU resource of the list
        sADMList->SetMenuID(menuListRef, 700);
        //     CheckPastePref(menuListRef);
    }
    sADMItem->SetTrackProc(menuItemRef, dialogPopupMenuTrackProc);
    // catch mouse down to do setup based on modifier keys
}
```

Once the dialog is created and all buttons and pop-up menu values are established, the example tracks the user's activity. This is done by using ADM Notify and ADM Tracker functions. In the above code, the following is used to set the Notify procedure as `dialogPopupMenuProc`:

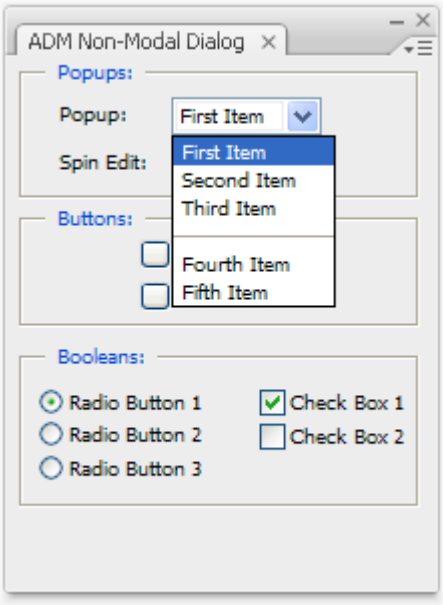
```
menuItemRef = sADMDialog->GetItem(dlg, kADMMenuItemID);
sADMItem->SetNotifyProc(menuItemRef, dialogPopupMenuProc);
```

This is covered later in this chapter. The following code is used to track user activity on this dialog item:

```
sADMItem->SetTrackProc(menuItemRef, dialogPopupMenuTrackProc);
```

## Pop-up list item

The first pop-up is a `kADMPopupListType` menu. It is shown in active position in the following figure.



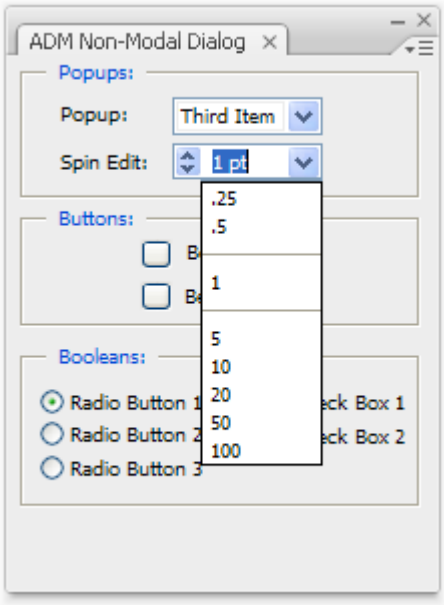

The following code in `DlgInit` creates and initializes the pop-up list:

```
// Create popup list
popupItemRef = sADMDialog->GetItem(dlg, kpopupItem);
sADMItem->SetNotifyProc(popupItemRef, PopupProc);
popupListRef = sADMItem->GetList(popupItemRef);
sADMList->SetMenuID(popupListRef, kpopupMenuID);

// Initialize popup list
entry = sADMList->GetEntry(popupListRef, 1);
    // The number you pass is 1-based not 0-based
sADMEntry->Select(entry, true);
```

Notice that `PopupProc` is used as a notify procedure.

## Spin-edit Item

Below the `PopupList` menu is a spin-edit text-edit item, shown activated in the following figure.




The code below from `DlgInit` creates and initializes the spin-edit pop-up list:

```
// Create SpinEditPopup list
spinEditPopupItemRef = sADMDialog->GetItem(dlg, kspinEditPopupItem);
sADMItem->SetNotifyProc(spinEditPopupItemRef, spinEditPopupProc);
spinEditPopupListRef = sADMItem->GetList(spinEditPopupItemRef);
sADMList->SetMenuID(spinEditPopupListRef, kspinEditPopuMenuID);

// Initialize SpinEditPopup list
entry = sADMList->GetEntry(spinEditPopupListRef, 4);
sADMItem->Invalidate(spinEditPopupItemRef);
sADMEntry->Select(entry, true);
```

Notice that `spinEditPopupProc` is used as a notify procedure.

## Radio-button and check-box items

At the bottom of the panel is an array of radio buttons and check boxes. ADM provides many useful automatic functions, including the ability to group radio buttons together so that within a group, selecting one button automatically de-selects the others. ADM considers any consecutively numbered radio button items as a group.

The radio and check-box buttons are initialized with the following code from `DlgInit`:

```
// All 3 radio button items will have the same notifier proc.
sADMItem->SetNotifyProc(sADMDialog->GetItem(dlg, kradioButton1Item), radioButtonProc);
sADMItem->SetNotifyProc(sADMDialog->GetItem(dlg, kradioButton2Item), radioButtonProc);
sADMItem->SetNotifyProc(sADMDialog->GetItem(dlg, kradioButton3Item), radioButtonProc);

// Initialize radio button group (items 7-9)
// Note: radio buttons with consecutive item numbers are automatically grouped by ADM
sADMItem->SetBooleanValue(sADMDialog->GetItem(dlg, kradioButton1Item), true);
sADMItem->SetBooleanValue(sADMDialog->GetItem(dlg, kradioButton2Item), false);
sADMItem->SetBooleanValue(sADMDialog->GetItem(dlg, kradioButton3Item), false);

/** Checkbox Items **/

// Each check box item will have its own notifier proc.
sADMItem->SetNotifyProc(sADMDialog->GetItem(dlg, kcheckBox1Item), checkBox1Proc);
sADMItem->SetNotifyProc(sADMDialog->GetItem(dlg, kcheckBox2Item), checkBox2Proc);

// Initialize checkboxes
sADMItem->SetBooleanValue(sADMDialog->GetItem(dlg, kcheckBox1Item), true);
sADMItem->SetBooleanValue(sADMDialog->GetItem(dlg, kcheckBox2Item), false);
```

## Button items

The `DlgInit` code sets up the button items (Beep and Beep Beep) with the code below:

```
// Attach the callbacks for the beep buttons
beepButtonItemRef = sADMDialog->GetItem(dlg, kbeepItem);
if (beepButtonItemRef)
{
       sADMItem->SetNotifyProc(beepButtonItemRef, beepButtonUp);
//     You could also do this stuff:
//     sADMItem->SetItemStyle(beepButtonItemRef,
//        kADMBlackRectPictureButtonStyle);
//     sADMItem->SetTrackProc(beepButtonItemRef, ButtonTrackProc);
//     sADMItem->SetPictureID(beepButtonItemRef, kADMDeleteEntryPictureID, NULL);
//     sADMItem->SetSelectedPictureID(beepButtonItemRef,
//        kADMDeleteEntryPictureID, NULL);
//     sADMItem->SetDisabledPictureID(beepButtonItemRef,
//        kADMDeleteEntryDisabledPictureID, NULL);
//     sADMItem->SetCursorID(beepButtonItemRef, pluginRef, kADMFingerCursorID,
//        NULL);
//     fxErr = sADMBasic->GetIndexString(
//     pluginRef, kTooltipStrings, trashButtonTipIndex, tipString, 62);
//     if (fxErr)
//        fxErr = kNoErr; // don't let lack of a tool tip stop the show
//     else if (tipString[0])
//        sADMItem->SetTipString(beepButtonItemRef, tipString);
}


beepBeepButtonItemRef = sADMDialog->GetItem(dlg, kbeepBeepItem);
if (beepBeepButtonItemRef)
{
    sADMItem->SetNotifyProc(beepBeepButtonItemRef, beepBeepButtonUp);
}
```

## Dialog positioning and docking

Unlike a modal dialog, a modeless or floating panel dialog "floats" above the host application and can be moved around the screen; therefore, there is additional overhead to position the dialog.

In addition, panel type dialogs can be combined or docked together into a single panel. For this arrangement, you must be concerned with the `positionCode` value. This sets the dialog's position within a docked/tabbed group. The following code from `DlgInit` takes care of this:

```
// Get the last known Docking group and Docking code out of the Prefs file
//sASLib->strcpy(groupName, kLayersPaletteDockGroup);
sAIPreference->GetStringPreference("ADMNonModalDialog",
    "kADM_DPDockGroupStr", groupName);
positionCode = 0x00001c00;     // Default: no dock, no tab group,
                               // front tab, zoom down, visible
sAIPreference->GetIntegerPreference("ADMNonModalDialog", "kADM_DPDockCodeStr",
    &positionCode);

// Pick a default location in case it has never come up before on this machine
sADMDialog->GetBoundsRect(dlg, &boundsRect);
sADMBasic->GetPaletteLayoutBounds(&dimensions);
location.h = dimensions.right - (boundsRect.right - boundsRect.left);
location.v = dimensions.bottom - (boundsRect.bottom - boundsRect.top);

// Get the last known location out of the Prefs file
sAIPreference->GetPointPreference("ADMNonModalDialog", "kADM_DPLocationStr",
    &location);

size.h = 208;    // Minimum width (which governs the inner client rect) + 2
//size.v = layerMinHeight;
size.v = 258;

#ifdef WIN_ENV          // Different rules about whether the borders and tabs
                        // are in the dlg rect
    size.v += 6;
    location.v -= 6;
    size.h += 4;
#endif
    // Get the last known size out of the Prefs file
    sAIPreference->GetPointPreference("ADMNonModalDialog", "kADM_DPSizeStr", &size);
    rect.left = location.h;
    rect.right = location.h + size.h;
    rect.top = location.v;
    rect.bottom = location.v + size.v;

    // Restore the size and location of the dialog
    sADMDialog->SetBoundsRect(dlg, &rect);
    // restore the position code of the dialog
    sADMDialogGroup->SetDialogGroupInfo(dlg, groupName, positionCode);

    // Initialize the panel internals
    //result = LayersDlgInit(dlg);

    // Clean up the application context and return
    sAIAppContext->PopAppContext(AppContext);
    POP_GLOBALS

    return fxErr;
}
```

# Notification procedures

Since the dialog box must be notified when the user interacts with the various dialog items, the code uses notification procedures like `dialogPopupMenuProc` in `admHandler.cpp` to handle any activity:

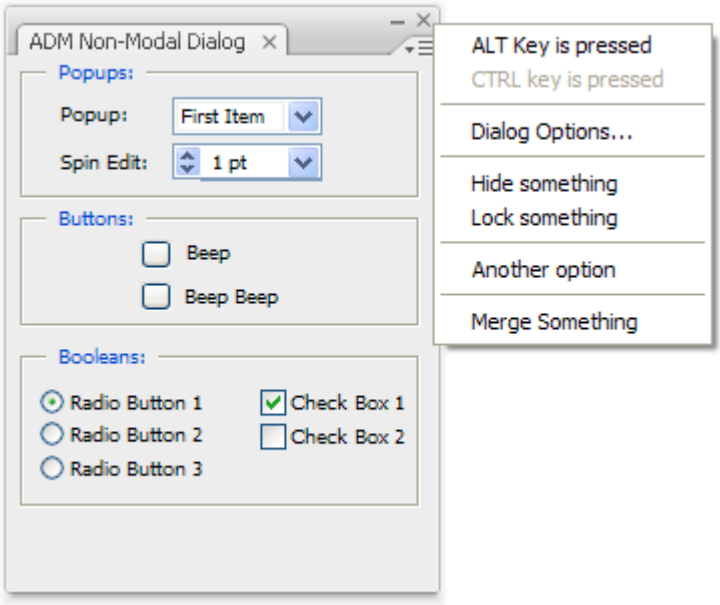
```
static void ASAPI dialogPopupMenuProc(ADMItemRef item,
    ADMNotifierRef notifier) {
    PUSH_GLOBALS

    int selection;
    // dispatch the notifier type
    if (sADMNotifier->IsNotifierType(notifier, kADMUserChangedNotifier) )
    selection = GetPopupSelection(item);
    POP_GLOBALS
}
```

This routine is passed the `ADMItemRef` and the notifier. Once the notifier is checked to make sure it is the right one (that is, `kADMUserChangedNotifier`), you get the selection by calling `GetPopupSelection`, as shown below:

```
static int GetPopupSelection(ADMItemRef item) {

    ADMListRef listRef;
    ADMEntryRef activeEntry;
    int  selection;

    listRef = sADMItem->GetList(item);
    // Get the current active entry
    activeEntry = sADMList->GetActiveEntry(listRef);
    // Get the index (0 based) of the entry
    selection = sADMEntry->GetIndex(activeEntry);

    return selection;

}
```

## Handling modifier keys

As noted above, when the user selects the right-arrow menu button shown again below, you can check whether a modifier key is pressed. If so, one of the first two items becomes selected; if not, they remain greyed out and de-selected. The figure below shows the flyout pop-up menu with the Alt key pressed while the right arrow was pressed.




This is handled with the `dialogPopupMenuTrackProc` procedure:

```
/* dialogPopupMenuTrackProc(), this is called at mouse down on the
popup menu. This is your opportunity to change the status of
menu items. In this example, the modifier keys enable menu items 1 and 2
*/

static ASBoolean ASAPI dialogPopupMenuTrackProc(ADMItemRef item,
    ADMTrackerRef tracker)
{
    PUSH_GLOBALS
    ADMAction           action;
    ASBoolean           doNotify;

    action = sADMTracker->GetAction(tracker);

    // Capture mouse down event
    if (action == kADMButtonDownAction)
    {
        SPPluginRef             pluginRef;
        ADMListRef              menuListRef;
        AIBoolean               commandOptionDown, commandControlDown;

        // Check if the (mac)OPTION (win)ALT modifier is pressed
        commandOptionDown = sADMTracker->TestModifier(tracker, kADMModKeyDownModifier);

        // Check if the (mac)COMMAND (win)CONTROL modifier is pressed
        commandControlDown = sADMTracker->TestModifier(tracker, kADMMenuKeyDownModifier);

        pluginRef = sADMItem->GetPluginRef(item);
        menuListRef = sADMItem->GetList(item);
```

```
            if (menuListRef)
            {
                // If command is pressed, enable the first menu item; otherwise, disable
                sADMEntry->Enable(sADMList->IndexEntry(menuListRef, 0), commandOptionDown);
                // If command is pressed, enable the second menu item; otherwise, disable
                sADMEntry->Enable(sADMList->IndexEntry(menuListRef, 1), commandControlDown);
            }
        }

        doNotify = sADMItem->DefaultTrack(item, tracker);

        POP_GLOBALS

        return doNotify;
    }
```

# For more information

For more information, see the ADMNonModalDialog sample plug-in and *Adobe Illustrator CS3 SDK API Reference*.

# 4 ADM Suites

This chapter briefly describes the following suites:



## Basic suite

The ADM Basic suite provides four types of functions:

- *Resource functions* — APIs that deal with platform resources.
- *User-interface functions* — APIs that deal with the GUI. User-interface functions control basic user communications like beeps, alerts, and tool tips.
- *Utility functions* — APIs for miscellaneous tasks.
- *Contextual-menu functions* — APIs used to create pop-up contextual menus. Contextual menus can be used to display a menu when the cursor is over a particular location and a combination of mouse states or modifier keys can be used to select an option. For example, pressing and holding down the mouse in an Internet browser window opens a pop-up menu with Back and Forward options. Most likely, these functions are used by the host application rather than a plug-in. Contextual menus must be destroyed when done.

**NOTE:** For user-interface functions, tool tips provide information about the ADM item pointed to by the mouse cursor. A predefined string describing the item appears to the right of the item after the mouse is positioned over it for a few seconds. When the mouse is moved, the tool tip disappears. The strings to use for a dialog are defined with `sADMDialog->LoadToolTips`. For Illustrator and other Adobe tools, the tool title is used for the tool tip.

### Accessing the suite

Acquire this suite by calling `SPBasicSuite::AcquireSuite()`. To determine the values to use for the name and version parameters in this call:

- For Illustrator, see ADMBasicSuite in *Adobe Illustrator CS3 SDK API Reference*.
- For other products, see the `ADMBasic.h` header file.

### For more information

For Illustrator, see ADMBasicSuite in *Adobe Illustrator CS3 SDK API Reference*.

For other products, see the documentation in the `ADMBasic.h` header file.

## Dialog suite

The ADM Dialog suite allows you to create and access ADM Dialog objects. Many of the functions are common to all ADM objects, such as text-access functions. Others are unique to dialogs; for instance, setting minimum and maximum sizes for resizable dialogs.

### Accessing the suite

Acquire this suite by calling `SPBasicSuite::AcquireSuite()`. To determine the values to use for the name and version parameters in this call:

- For Illustrator, see ADMDialogSuite in *Adobe Illustrator CS3 SDK API Reference*.
- For other products, see the `ADMDialog.h` header file.

### Dialog styles

ADM supports several types of dialog styles. Valid modal-dialog styles are kADMModalDialogStyle, kADMAlertDialogStyle, and kADMSystemAlertDialogStyle. Each dialog type is assigned a value. Alert styles (`0`, `1`, `8`) are only for modal dialogs; all others are for non-modal dialogs. Style constants above FFFF are reserved for host-application use. For details, see ADMDialogSuite in *Adobe Illustrator CS3 SDK API Reference* and the `ADMDialog.h` header file.

For more complete descriptions of the dialog styles, see Chapter 1, "ADM Overview."

## Standard dialog-item IDs

ADM provides many types of dialog items, as outlined in Chapter 1, "ADM Overview. Each standard dialog item is assigned an ID. For details, see ADMDialogSuite in *Adobe Illustrator CS3 SDK API Reference* and the `ADMDialog.h` header file.

## Callbacks

If default operation is not desired, most ADM dialog items support programmer-supplied callbacks, listed below and defined in `ADMDialog.h`. As shown below, these procedures include the dialog Init proc, Draw proc, Tracker proc, Notify proc, and dialog Destroy proc. The timer and timer-abort procedures are used to attach timers to dialogs.

```
ADMDialogInitProc
ADMDialogDrawProc
ADMDialogTrackProc
ADMDialogNotifyProc
ADMDialogDestroyProc
ADMDialogTimerProc
ADMDialogTimerAbortProc
```

## ADM help support

ADM has built-in support for ASHelp, a WinHelp-type help system. ASHelp uses WinHelp file definitions in a cross-platform fashion. Every item has a helpID, and the system can operate in contextual fashion. For example, selecting Command ? in Mac OS or Alt + F1 in Windows lets you click an item and see its help resource. For plug-ins to support help files, there must be a Plugin Help location in the `PiPL` resource.

**CAUTION:** The Help APIs are deprecated.

## For more information

For Illustrator, see ADMDialogSuite in *Adobe Illustrator CS3 SDK API Reference*.

For other products, see the documentation in the `ADMDialog.h` header file.

# Dialog Group suite

The ADM Dialog Group suite handles the "grouping" or docking of dialogs, as in a docked panel window. It also provides access to the collection of all dialogs ADM knows about at any given time.

## Accessing the suite

Acquire this suite by calling `SPBasicSuite::AcquireSuite()`. To determine the values to use for the name and version parameters in this call:

- For Illustrator, see ADMDialogGroupSuite in *Adobe Illustrator CS3 SDK API Reference*.
- For other products, see the `ADMDialogGroup.h` header file.

## Position code and group name

The position code parameter, set and retrieved using the `sADMDialogGroup->SetDialogGroupInfo` and `sADMDialogGroup->GetDialogGroupInfo` functions, is used to restore a dialog's position within a docked/tabbed group. (Tabbed dialogs can be "floating," or they can be "docked" with other dialogs into a docked/tabbed group.) These functions also can be used to set and retrieve the group name. The dialog group is referred to using the name of the dialog that is the first tab in the top dock of the group.

When docking several panels, it is necessary to determine which panel is first, second, third, etc., and where the tab is located (first, second, etc.). All these settings are determined by the position code. For a description of the values that position code can take, see `ADMDialogGroup.h`.

## For more information

For Illustrator, see ADMDialogGroupSuite in *Adobe Illustrator CS3 SDK API Reference*.

For other products, see the documentation in the `ADMDialogGroup.h.h` header file.

# Drawer suite

The ADM Drawer suite is a set of cross-platform imaging functions for use within custom ADM drawer functions. If you create a new type of dialog item or want to embellish an existing one, use the ADM Drawer suite functions to create its appearance.

The functions of the ADM Drawer suite are similar to most platform imaging APIs, but they are optimized for user-interface work. The suite includes basic imaging (e.g., `sADMDrawer->DrawLine`, `sADMDrawer->DrawRect`, and `sADMDrawer->SetClipRect`) and text-handling functions. The color system provides an easy way to specify user interface colors; for instance, two ADM color constants are `kADMDisabledColor` and `kADMButtonDownColor`. Similarly, the fonts available to the text functions are limited to those needed to make dialog items, like `kADMItalicPaletteFont`. In addition, functions are provided that simplify the implementation of many standard dialog items, like `sADMDrawer->DrawIcon` and `sADMDrawer->DrawUpArrow`. If the drawing capabilities of the ADM Drawer suite are insufficient, you can access a drawing port for using platform imaging functions (or, for internal Adobe development, Adobe's imaging functions).

## Accessing the suite

Acquire this suite by calling `SPBasicSuite::AcquireSuite()`. To determine the values to use for the name and version parameters in this call:

- For Illustrator, see ADMDrawerSuite in *Adobe Illustrator CS3 SDK API Reference*.
- For other products, see the `ADMDrawer.h` header file.

## Drawer functions

ADM drawers are callback functions assigned to ADM objects. Their purpose is to draw on the screen the object to which they are assigned. To specify an ADM drawer function to use with an ADM object, use the assignment functions:

```
sADMDialog->SetDrawProc
sADMItem->SetDrawProc
sADMList->SetDrawProc
sADMHierarchyList->SetDrawProc
```

For ADM dialogs and ADM items, the assignment function is found in the object suite. Drawer functions for ADM Entry objects are assigned to the list that contains them. All entries in a list have the same drawer function.

All ADM drawer callbacks are defined similarly:

```
typedef void ASAPI (*ADMObjectDrawProc)(ADMObjectRef inObject,
       ADMDrawerRef inDrawer);
```

*Object* is a reference to the dialog, item, or entry that is to be drawn. The `ADMDrawerRef` argument basically is a graphics-device reference and is used with the functions in this suite to indicate the context for the imaging operation.

All ADM objects have a default drawer function that provides their normal appearance. If your customization to an object is an embellishment of this standard appearance, you can call the default drawer and then modify it (the default drawer for ADM User items does nothing). To call the default drawer, use a function of the object suite:

```
sADMDialog->DefaultDraw
sADMItem->DefaultDraw
sADMListEntry->DefaultDraw
sADMListEntry->DefaultDraw
```

Pass the `DefaultDraw` function the arguments that were passed to your customer drawer function; for instance:

```
void mySquareDrawHandler(ADMItemRef item, ADMDrawerRef drawer) {
       sADMItem->DefaultDraw(item, drawer);
}
```

## Using drawer functions

The functions in the ADM Drawer suite require an `ADMDrawerRef`, which is the target for the drawing operation. One of the arguments passed to your drawer function is a drawer reference, and it is simply passed to the each ADM drawer function:

```
void mySquareDrawHandler(ADMItemRef item, ADMDrawerRef drawer) {
       ASRect boundsRect;
       sADMItem->DefaultDraw(item, drawer);

       sADMDrawer->GetBoundsRect(drawer, &boundsRect);
       boundsRect.top -= 2;
       boundsRect.bottom += 2;
       boundsRect.left -= 2;
       boundsRect.right += 2;

       sADMDrawer->SetADMColor(drawer, kADMShadowColor)
       sADMDrawer->DrawRect(drawer, &boundsRect)
}
```

## Fonts and colors

The ADM Drawer suite has a streamlined model for colors and fonts that offers benefits like facilitating the design of user-interface components with the Adobe "look" and simplifying conformance to platform standards. Both these qualities help the user have a consistent experience with Adobe applications and their plug-ins.

The normal colors you use in implementing a dialog item have been abstracted. Except for black and white, the keywords for the colors indicate the role they play in the user interface. Some of the constants may represent the same color. In addition to helping provide a consistent user experience, another benefit of using the ADM color constants is that your interface adapts to the changing platform interface standards as ADM does.The standard colors used by ADM are listed in `theADMColor struct` in `ADMTypes.h`.

Fonts are simplified from the hundreds that are potentially available to just a few, abstracted so that the correct font is used on a given platform. The standard fonts used by ADM are listed in `ADMFont struct` in `ADMTypes.h`.

The constant names for the fonts indicate their purpose. For instance, the `kADMDialogFont` font generally is bigger than the `kADMPaletteFont` font and is used for modal dialogs. Floating windows should use the smaller font to reduce the screen area needed by a window that always is present. Font attributes like size are handled automatically by ADM. ADM uses any required system settings when setting these attributes.

## Drawer coordinate space

All drawing is done in coordinates local to the object being drawn, specified relative to the drawer origin. By default, the origin of the drawer is the top-left corner of its object's bounding rectangle. You can redefine the origin if desired, and doing so may simplify some drawing operations.

Pixels are drawn in a Mac OS-like fashion, where a pixel is drawn down and to the right of the indicated coordinate.

## Drawing modes

ADM provides two drawing modes, *normal mode* and *XOR mode*, that affect how drawing operations occur (see `ADMDrawer.h`). Valid styles are kADMNormalMode, kADMXORMode, and kADMDummyMode.

In normal mode, a graphics operation overwrites the background entirely. In XOR mode, the background color is inverted when the color of the pixel being drawn is black. This is illustrated in the following figure.

A line drawn in kADMNormalMode

A line drawn in kADMXORMode

Graphics commands also are affected by a *clipping path*. A clipping path is a defined region outside of which graphic operations have no affect. The figure below shows drawing with and without a clipping path.

Horizontal lines

Horizontal lines
with a clipping path

Clipping paths can be set in several ways: as a rectangle, as a polygon, and by combining multiple rectangles and polygons.

### For more information

For Illustrator, see ADMDrawerSuite in *Adobe Illustrator CS3 SDK API Reference*.

For other products, see the documentation in the `ADMDrawer.h` header file.

## Entry suite

The ADM Entry suite allows you to create and access ADM Entry objects that are used in conjunction with the ADM List suite. Most of the functions are common to all ADM objects, like text-access functions. A few are similar to ADM Item objects, like setting picture IDs. The remainder are unique to ADM Entry objects, for instance, checking if an entry is selected. This function reference builds on ideas established in Chapter 1, "ADM Overview.

### Accessing the suite

Acquire this suite by calling `SPBasicSuite::AcquireSuite()`. To determine the values to use for the name and version parameters in this call:

- For Illustrator, see ADMEntrySuite in *Adobe Illustrator CS3 SDK API Reference*.
- For other products, see the `ADMEntry.h` header file.

### Initializing an entry

When you assign a menu ID to a list, ADM creates and initializes the list's ADM Entry objects. They are given an ID and assigned a text string. If you need initialization beyond this, or if you create your own ADM entries, you need to perform these tasks yourself. This can be handled by assigning an `ADMEntryInitProc` (see `ADMList.h`) callback function to the list. It is called for each entry that is created.

What you need to initialize depends on what the entries represent. ADM has several standard entry properties that can be set, such as picture IDs. In addition, you can perform your own initialization, like allocating memory or loading resources. If you do your own initialization, you probably will need to also replace the list entries' draw function.

**NOTE:** Unlike ADM dialogs and items, custom handler functions for entries, like Init and Draw procs, are set for the containing list, not for individual entries. (See "Custom lists" on page 95.)

### Help support

ADM has built-in support for ASHelp, a WinHelp-type help system. ASHelp uses WinHelp file definitions in a cross-platform fashion. Every item has a helpID, and the system can operate in contextual fashion. For example, selecting Command ? in Mac OS or Alt + F1 in Windows lets you click an item and see its help resource. For plug-ins to support help files, there must be a Plugin Help location in the `PiPL` resource.

**CAUTION:** The Help APIs are deprecated.

### For more information

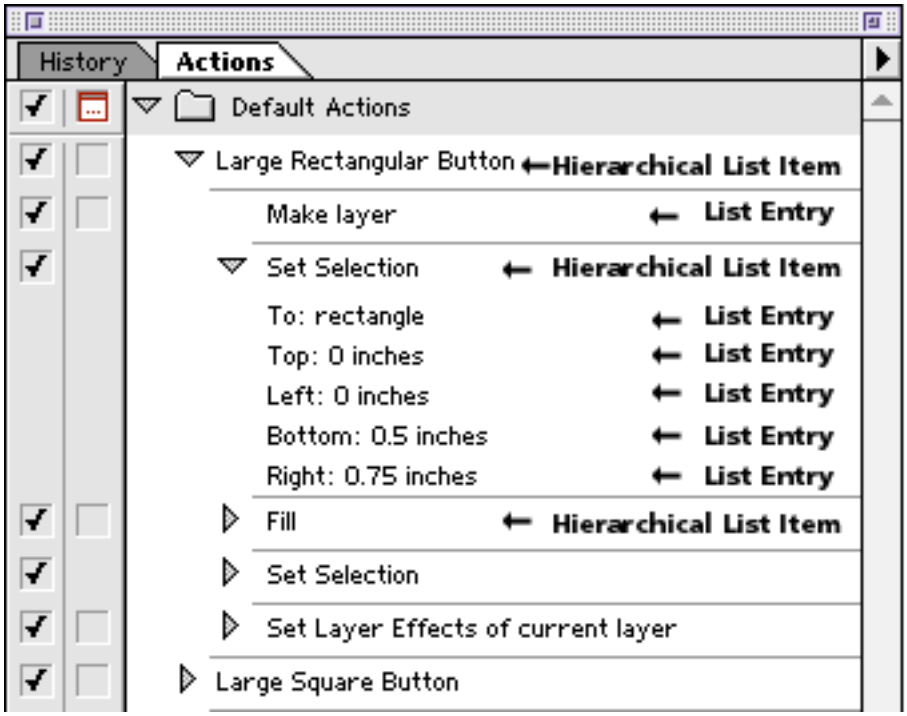
For Illustrator, see ADMEntrySuite in *Adobe Illustrator CS3 SDK API Reference*.

For other products, see the documentation in the `ADMEntry.h` header file.

## Hierarchy List suite

The ADM Hierarchy List suite allows you to access ADM Hierarchy List objects and ADM List Entry objects. Since an ADM Hierarchy List object is an extended property of a standard ADM Item object, this suite lacks many of the functions common to ADM objects; however, you can access a hierarchy list's ADM item and do common operations on it. Using functions in this suite, you can initialize the hierarchy list, and you can create, destroy, customize, and iterate through the ADM list entries of a hierarchy list. The Hierarchy List suite is used in conjunction with the ADM List Entry suite to further access list-related information.

**NOTE:** The relationship between ADM Hierarchy List objects and ADM List Entry objects is the same as that between ADM List objects and ADM Entry objects; that is, list entries are the elements of a hierarchy list. List entries themselves may be hierarchy lists with list-entry children of their own. See the figure below.




## Accessing the suite

Acquire this suite by calling `SPBasicSuite::AcquireSuite()`. To determine the values to use for the name and version parameters in this call:

- For Illustrator, see ADMHierarchyListSuite in *Adobe Illustrator CS3 SDK API Reference*.
- For other products, see the `ADMHierarchyList.h` header file.

## Hierarchy lists and list entries

ADM Hierarchy List objects are used by ADM Item objects to provide a list of expandable choices, including list boxes, pop-up lists, pop-up menus, spin-edit popups, and text-edit popups. An ADM hierarchy list comprises ADM list entries.

ADM hierarchy lists do not have many standard properties, such as a plug-in and bounds. Rather, these are defined using the ADM hierarchy list's item. To access them, use `sADMHierarchyList->GetItem` to get the item owning the list, then use the ADM Item suite functions with the returned item reference.

ADM hierarchy lists have special properties, like a menu resource ID and a group of list entries. ADM list entries have added properties, including an index and a selected state. These entry properties are used by the ADM Hierarchy List suite to access the entries. The index is the position of the entry in list. The selected state indicates the user has selected the item (others may be selected in the case of a multi-select list).

## Using the Hierarchy List suite

To get the hierarchy list object for a list entry, use the `sADMListEntry->GetItem` function:

```
ADMHierarchyListRef theHierarchyList = sADMListEntry->GetList(theListEntry);
```

Once this is done you can use the ADM Hierarchy suite functions.

To initialize a list, assign it a menu resource ID:

```
sADMHierarchyList->SetMenuID(theItemsList, gPlugInRef, 16000, "Choices");
```

You also can create each entry with the ADM Hierarchy List suite's `sADMHierarchyList->InsertEntry` function; for instance:

```
for (index = 0; index < kNumberEntries; index++)
{
    char menuText[255];
    ADMListEntryRef entry = sADMHierarchyList-
>InsertEntry(theItemList, index);
    sADMBasic->GetIndexString(thePlugin, 16000, index, menuText, 255);
    sADMListEntry->SetText(entry, menuText);
    sADMListEntry->SetID(entry, index);
}
```

Note that list indices are 0-based.

To get the currently selected item of a single selection list, use the `sADMHierarchyList->GetActiveEntry` function, then get the entry's index:

```
ASInt32 GetHierarchyListValue(ADMItem theListItem)
{
    ADMHierarchyListRef theList = sADMItem->GetList(theListItem);
    ADMListEntryRef theEntry = sADMHierarchyList->GetActiveEntry(theList);
    return sADMListEntry->GetID(theEntry);
}
```

To get each selected item in a multiple-selection list, get the selection count and iterate through the selections:

```
ASInt32 count = sADMHierarchyList->NumberOfSelectedEntries(theList);
for (index = 0; index < count; index++)
{
    ADMListEntryRef entry = sADMHierarchyList->IndexSelectedEntry(theList, index);
    doSomethingToSelectedEntry(theEntry);
}
```

## Custom hierarchy lists

You can customize an ADM Hierarchy List object just like other ADM items. This is done by defining one or more event-handler functions. Because ADM hierarchy lists are closely linked to ADM list entries, the process is slightly different.

The ADM hierarchy list does not have its own event-handler functions. To do something to the list as a whole in a handler, set the handler function for the list; for instance, to annotate the list, set the drawer function for the list. These are assigned using the ADM Item suite.

To change the behavior of the hierarchy list at a lower level, set the handler functions of the list's entries; for instance, to change how each list entry draws, set the drawer function for the list's entries. This is done at the hierarchy-list level, using the ADM Hierarchy List suite, and it affects all list entries in a list. You cannot directly set a handler function for an individual list entry; a custom handler function for a hierarchy list must work for all its list entries.

To use the default behavior for a hierarchy list item, you use the ADM Item suite functions. To use the default behavior for a list entry, use functions in the ADM List Entry suite, not the ADM Hierarchy List suite.

### For more information

For Illustrator, see ADMHierarchyListSuite in *Adobe Illustrator CS3 SDK API Reference*.

For other products, see the documentation in the `ADMHierarchyList.h` header file.

## Icon suite

The ADM Icon suite provides a standard interface to picture resources on multiple platforms. The suite supports pictures and icons on Windows and Mac OS and refers to them as ADM icons.

### Accessing the suite

Acquire this suite by calling `SPBasicSuite::AcquireSuite()`. To determine the values to use for the name and version parameters in this call:

- For Illustrator, see ADMIconSuite in *Adobe Illustrator CS3 SDK API Reference*.
- For other products, see the `ADMIcon.h` header file.

### Icons

ADM provides a generic icon interface to several platform-based resource types. When an icon is read from a plug-in file, ADM searches the supported resource types for the given resource ID. The resource is read into memory, and a reference to the ADM icon is returned to the caller.

The supported resource types are defined by the ADMIconType enumeration. Resources are read using the `sADMIconSuite->GetFromResource` function.

```
typedef enum
{
    // Mac types
    kCICN, kPICT, kIconSuite,
    // Windows types
    kWinIcon, kBMP,
    // Either type
    kADMImageIcon,
    kUnknown
} ADMIconType;
```

Icons of type `kIconSuite` and `kWinIcon` can have multiple icons with multiple depths, but all the supplied icons should have the same dimensions. In Mac OS, icon-suite resources are searched in the following order:

Large (ICN#/icl4/icl8)

Small (ics#/ics4/ics8)

Mini (icm#/ics4/ics8)

**NOTE:** In Mac OS, the `CICN` resource is provided for backward compatibility with Adobe Illustrator 6.0. Resources of this type are not part of the resource search. Icon suites are the preferred format.

To draw an ADM icon, use the `sADMDrawer->DrawIcon` and `sADMDrawer->DrawIconCentered` functions.

### For more information

For Illustrator, see ADMIconSuite in *Adobe Illustrator CS3 SDK API Reference*.

For other products, see the documentation in the `ADMIcon.h` header file.

## Image suite

The ADM Image suite provides a means for creating off-screen images that can be displayed and manipulated with ADM Drawer suite functions (see "Drawer suite" on page 80).

### Accessing the suite

Acquire this suite by calling `SPBasicSuite::AcquireSuite()`. To determine the values to use for the name and version parameters in this call:

- For Illustrator, see ADMImageSuite in *Adobe Illustrator CS3 SDK API Reference*.
- For other products, see the `ADMImage.h` header file.

### For more information

For Illustrator, see ADMImageSuite in *Adobe Illustrator CS3 SDK API Reference*.

For other products, see the documentation in the `ADMImage.h` header file.

## Item suite

The ADM Item suite allows you to create and access ADM Item objects. Many of the functions are those common to all ADM objects, like text-access functions. Others are unique to dialog items; for instance, setting text item numerics and picture IDs. This function reference builds on ideas established in Chapter 1, "ADM Overview.

### Accessing the suite

Acquire this suite by calling `SPBasicSuite::AcquireSuite()`. To determine the values to use for the name and version parameters in this call:

- For Illustrator, see ADMItemSuite in *Adobe Illustrator CS3 SDK API Reference*.
- For other products, see the `ADMItem.h` header file.

## Initializing ADM items

When you create a modal or non-modal ADM dialog, ADM automatically creates ADM items according to the dialog's item-list resource. These items have the default initialization. This means minimal initialization, so the dialog initialization function you provide needs to further initialize the items.

Exactly what you need to initialize depends on the item type. You may want to set an item style if it cannot be set from the resource. All items can be enabled or disabled. You may want to make one edit-text item active. Also, any item that needs to interact with other items should have a notification function. Item characteristics you might want to initialize for a given item are listed in the following table.

| **Item type (see `ADMItem.h`)** | **Initialization** |
|---|---|
| `kADMChasingArrowsType` | Mac OS only. This typically would be used transiently to indicate the user needs to wait. No initialization is required. |
| `kADMDialType` | Set the dial's minimum, maximum, and current value. Assign a notification proc. |
| `kADMFrameType` | There is likely nothing to initialize. You can set the style in the resource. You may want to set the frame text if it has a title. |
| `kADMHierarchyListBoxType` | Get the item's list and set its menu ID. If necessary, activate an entry or entries. Assign initialization, notification, and drawer functions for the entries. |
| `kADMItemGroupType` | Nothing needs to be done in an item group's InitProc. At the time the Init proc is called, no child items have been added, so there is no way to apply any properties to them as a group. Since previously assigned item-group properties are not propagated to child items when they are added, there is no point setting any of those values before all the children are added. Perform the following steps to create an item group:<br>➤ s`ADMItem->Create` the item group (this causes the InitProc to be called before returning from `sADMItem->Create`).<br>➤ Add any child items that need to be added to this group (use `sADMItem->AddItem`).<br>➤ Set any properties for the group that should be propagated to the children (using the various ADM Item suite Set mutators). |
| `kADMListBoxType` | Get the item's list and set its menu ID. If necessary, activate an entry or entries. Assign a notification function for the entries. |
| `kADMPasswordTextEditType` | Somewhat like `kADMTextEditType`, except passwords typically are not numeric and do not have default values. Set the maximum length in addition to the usual (enable/disable and set Notify proc). |

| Item type (see ADMItem.h) | Initialization |
|---|---|
| kADMPictureCheckBoxType | Set the picture ID. You may want to disable and select picture IDs as well. It might or might not need a notification function. |
| kADMPicturePushButtonType | Set the picture ID. You may want to disable and select picture IDs as well. Also, a push button needs a notification function. |
| kADMPictureRadioButtonType | Set the picture ID. You may want to disable and select picture IDs as well. Set the boolean value of one radio button in a group to be true. You do not need to use a notification routine to handle radio-button groups if all buttons in the group have consecutive IDs; ADM handles this case automatically. |
| kADMPictureStaticType | Set the picture ID. You may want to disable and select picture IDs as well. You may want a notification for an easter egg. |
| kADMPopupControlButtonType | Set the usual settings (enable/disable and set Notify proc). By default, this type of item has a pop-up slider, so you set the min and max values to specify the range of the whole item. See Set***Value functions. |
| kADMPopupControlType | Set the usual settings (enable/disable and set Notify proc). By default, this type of item has a pop-up slider, so you set the min and max values to specify the range of the whole item. See Set***Value functions. |
| kADMPopupListType | Get the item's list and set its menu ID. If necessary, activate an entry or entries. |
| kADMPopupMenuType | Get the item's list and set its menu ID. If necessary, activate an entry or entries. |
| kADMPopupSpinEditControlType | Set the usual settings (enable/disable and set Notify proc). Set the item's text. These types of items can have only numeric data (floats or ints, negative values are acceptable). Use sADMItem->SetText or Set***Value functions. By default, this type of item has a pop-up slider, so you set the min and max values to specify the range of the whole item. |
| kADMProgressBarType | Set the initial value. This item type typically exists transiently while an operation is being performed. |
| kADMResizeType | Set a notification function for the item. |
| kADMScrollbarType | Set the item's range and value. Set the large and small increments. Assign notification and tracking functions. Set the item's draw function for graphic feedback (e.g., a current picture). |
| kADMScrollingPopupListType | Get the item's list and set its menu ID. If necessary, activate an entry or entries. |

| Item type (see ADMItem.h) | Initialization |
|---|---|
| kADMSliderType | Set the item's range and value. Assign a notification function if necessary. Set the item's draw function if it has graphic feedback (e.g., a color slider). |
| kADMSpinEditPopupType | Get the item's list and set its menu ID. If necessary, activate an entry or entries. Set the item's text. Set any numerics, including ranges, units, etc. Set the small increment to be used for the spinner. |
| kADMSpinEditScrollingPopupType | Get the item's list and set its menu ID. If necessary, activate an entry or entries. Set the item's text. Set any numerics, including ranges, units, etc. Set the small increment to be used for the spinner. |
| kADMSpinEditType | Set the item's text. Set any numerics, including ranges, units, etc. Set the increments to be used for the spinner. |
| kADMTabbedMenuType | Obsolete. |
| kADMTextCheckBoxType | Set the item's boolean value. |
| kADMTextEditMultiLineReadOnlyType | Same as kADMTextEditMultiLineType, except you cannot type in them or select them. |
| kADMTextEditMultiLineType | Set the item's text and maximum text length. Set any numerics, including ranges, units, and precision. |
| kADMTextEditPopupType | Get the item's list and set its menu ID. If necessary, activate an entry or entries. Set the item's text. Set any numerics, including ranges, units, and precision. |
| kADMTextEditReadOnlyType | Same as kADMTextEditType, except you cannot type in them or select them. |
| kADMTextEditScrollingPopupType | Get the item's list and set its menu ID. If necessary, activate an entry or entries. Set the item's text. Set any numerics, including ranges, units, and precision. |
| kADMTextEditType | Set the item's text and maximum text length. Set any numerics, including ranges, units, and precision. |
| kADMTextPushButtonType | Push buttons need a notification routine. |
| kADMTextRadioButtonType | Set the boolean value of one radio button in a group to be true. You do not need to use a notification routine to handle radio-button groups if all buttons in the group have consecutive IDs; ADM handles this case automatically. |
| kADMTextStaticMultilineType | Possibly enable or disable. |

| Item type (see ADMItem.h) | Initialization |
|---|---|
| kADMTextStaticType | Possibly enable or disable. |
| kADMUserType | Since this usually is used for custom items, you set the draw, notification, and tracking functions. User items have the same properties as any other items, so you also may want to set a picture, text, or value. You determine the needed initialization. |

## FloatToText and TextToFloat functions

The ADM ADMItemTextToFloatProc/ADMItemFloatToTextProc (see ADMItem.h) routines are available to plug-in programmers who want to override the ADM default text-to-float and float-to-text routine behaviors. You can override them by using sADMItem->SetTextToFloatProc and sADMItem->SetFloatToTextProc. With ADMItemFloatToTextProc, you can affect the float value used for the item; reciprocally, with ADMItemTextToFloatProc, you can affect the text that is made visible to the user.

For instance, these would be used when you have an ADM numeric-text item that has a min and max but also can have no value to indicate it is unused. Under the default ADM text-to-float routine, when the user deletes the value, ADM puts a 0 back in the field. To keep it from doing this, use ADMItemTextToFloatProc.

If the ADMItemTextToFloatProc returns false, the text is assumed to be invalid, and a notification is presented to the user. If true is returned, and the item is known (see sADMItem->IsKnown), ADM checks it against the min and max values and acts accordingly. If true is returned and your ADMItemTextToFloatProc marked the value as unknown, the text is used as is, and no notification to the user is made.

The float-to-text functions are as follows:

- sADMItem->DefaultFloatToText
- sADMItem->GetFloatToTextProc
- sADMItem->SetFloatToTextProc
- sADMItem->DefaultTextToFloat
- sADMItem->GetTextToFloatProc
- sADMItem->SetTextToFloatProc

## Help support

**CAUTION:** The Help API is deprecated. This does *not* include tool tips.

ADM has built-in support for ASHelp, a WinHelp-type help system. This allows ADM user interfaces to use WinHelp-compatible files to provide assistance to the user. ASHelp uses WinHelp file definitions in a cross-platform fashion. Each ADM object can have a help ID that is used to identify a location in the help file to be displayed when help is triggered. The system can operate in contextual fashion. For example, selecting Command ? in Mac OS or Alt + F1 in Windows lets you click an item and see its help resource. For plug-ins to support help files, there must be a Plugin Help location in the PiPL resource.

The host application probably provides a help file for itself and all the plug-ins that ship with it. Individual plug-ins also can provide their own help files independently of this.

To specify an alternate help file, a property is created in the plug-in's PiPL. If this property does not exist, help for the plug-in is assumed to be in the main help file. The property is as follows:

```
#define kHelpFileStrIDProperty  'HlpS'
```

The data for the property is versioned with the current specification:

```
//current is version 0
typedef struct PIHelpFileDesc
{
       long fVersion;
       long fFileNameStrID;
} PIHelpFileDesc
```

The fFileNameStrID is an indexed string resource giving the name of the help file to use for the plug-in. The name should be the first string in the list:

```
#define kHelpNativeStrIndex                     1
```

So, on Mac OS, in addition to the PiPL/property, a 'STR#' resource is created with the indicated ID and the name of the help file as the first (and possibly only) string. On Windows, a string resource with the ID "fFileNameStrID + 1" identifies the help file.

The plug-in help file must be in the same directory as the main application help file.

The ADM help-support functions are as follows:

- sADMItem->SetHelpID
- sADMItem->GetHelpID
- sADMItem->Help
- sADMItem->SetTipString
- sADMItem->GetTipString
- sADMItem->GetTipStringLength
- sADMItem->EnableTip
- sADMItem->IsTipEnabled
- sADMItem->ShowToolTip
- sADMItem->HideToolTip

### For more information

For Illustrator, see ADMItemSuite in *Adobe Illustrator CS3 SDK API Reference*.

For other products, see the documentation in the ADMItem.h header file.

## List suite

The ADM List suite allows you to access ADM List objects and ADM List entries. Since an ADM list is an extended property of a standard ADM item, this suite lacks many of the functions common to ADM objects; however, you can access the list's ADM Item and do common operations on it. Using functions in

this suite, you can initialize the list, and you can create, destroy, customize, and iterate through the entries of a list. The list suite is used in conjunction with the ADM Entry suite to further access list-related information.

## Accessing the suite

Acquire this suite by calling `SPBasicSuite::AcquireSuite()`. To determine the values to use for the name and version parameters in this call:

- For Illustrator, see ADMListSuite in *Adobe Illustrator CS3 SDK API Reference*.
- For other products, see the `ADMList.h` header file.

## Lists and entries

ADM lists are used by any ADM item that provides a list of choices, including list boxes, pop-up lists, pop-up menus, spin-edit popups, and text-edit popups. An ADM list is composed of ADM entries.

ADM lists do not have many standard properties, like plug-in and bounds; rather, these are defined using the ADM list's item. To access them, use the `sADMList->GetItem` function to get the item owning the list, then use the ADM Item suite functions (see "Item suite" on page 88") with the returned-item reference.

ADM lists have special properties, such as a menu-resource ID and a group of entries. ADM entries (see "Entry suite" on page 83") have additional properties, including an index and a selected state. These entry properties are used by the ADM List suite to access the entries. The index is the position of the entry in list. The selected state indicates the user has selected the item (others may be selected in the case of a multi-select list).

## Using the List suite

To get the ADM List object for an item, use `sADMItem->GetList`:

```
ADMListRef theItemsList = sADMItem->GetList(theItem);
```

Once this is done, you can use the ADM List and Entry suite functions to modify it.

To initialize a list, assign it a menu-resource ID:

```
sADMList->SetMenuID(theItemsList, gPlugInRef, 16000, "Choices");
```

You also can create each entry with the `sADMList->InsertEntry` function followed by the `sADMEntry->SetText` and `sADMEntry->SetID` functions:

```
for (index = 0; index < kNumberEntries; index++)
{
    char menuText[255];
    ADMEntryRef entry = sADMList->InsertEntry(theItemList, index);
    sADMBasic->GetIndexString(thePlugin, 16000, index, menuText, 255);
    sADMEntry->SetText(entry, menuText);
    sADMEntry->SetID(entry, index);
}
```

**NOTE:** List indices are 0-based.

To get the currently selected item of a single selection list, use the `sADMList->GetActiveEntry` function, then get the entry's index, as follows:

```
int GetListValue(ADMItem theListItem)
{
    ADMListRef theList = sADMItem->GetList(theListItem);
    ADMListEntryRef theEntry = sADMList->GetActiveEntry(theItemList);
    return sADMEntry->GetID(theEntry);
}
```

To get each selected item in a multiple selection list, get the selection count and iterate through the selections, as follows:

```
int count = sADMList->NumberOfSelectedEntries(theList);
for (index = 0; index < count; index++)
{
    ADMEntryRef entry = sADMList->IndexSelectedEntry(theList, index);
    doSomethingToSelectedEntry(theEntry);
}
```

### Custom lists

You can customize an ADM List object just as you can customize other ADM items. This is done by defining one or more event-handler functions. Because ADM lists are closely linked to ADM entries, the process is slightly different.

The ADM list does not have its own event-handler functions. To do something to the list as a whole in a handler, set the handler function for the list; for instance, to annotate the list, set the drawer function for the list. These are assigned using the ADM Item suite.

To change the behavior of the list at a lower level, set the handler functions of the list's entries; for instance, to change how each entry draws, set the drawer function for the list's entries. This is done at the list level using the ADM List suite, and it affects all entries in a list. You cannot directly set a handler function for an individual entry; a custom handler function for a list must work for all its entries.

To use the default behavior for a list item, use the ADM Item suite functions. To use the default behavior for a list entry, use functions in the ADM Entry suite; not the ADM List suite.

### For more information

For Illustrator, see ADMListSuite in *Adobe Illustrator CS3 SDK API Reference*.

For other products, see the documentation in the `ADMList.h` header file.

## List Entry suite

An ADM List Entry object is a member of an ADM Hierarchical List object. It acts exactly like an ADM Entry, except it has the possibility of containing child lists and entries. Since an ADM list entry is an extended property of a standard ADM Item, you can access the list entry's ADM item and perform operations on it. Using functions in this suite, you can create, destroy, customize, and iterate through the entries of a hierarchical list. The List Entry suite is used in conjunction with the ADM Hierarchical List suite to further access list-related information.

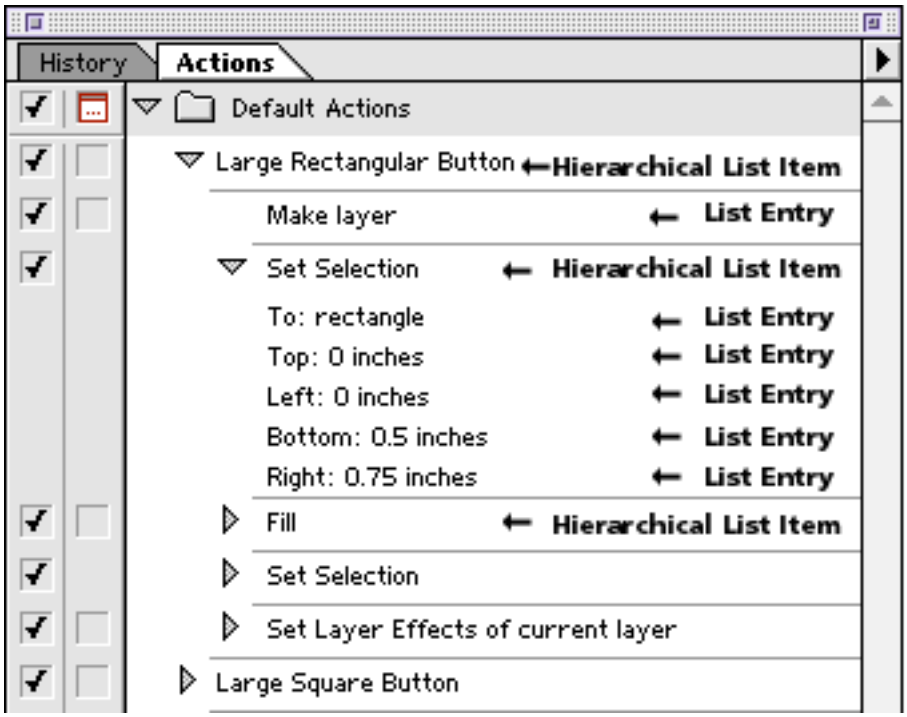
## Accessing the suite

Acquire this suite by calling `SPBasicSuite::AcquireSuite()`. To determine the values to use for the name and version parameters in this call:

- For Illustrator, see ADMListEntrySuite in *Adobe Illustrator CS3 SDK API Reference*.
- For other products, see the `ADMListEntry.h` header file.

## List objects and entries

ADM List objects are used by any ADM Item object that provides a list of choices, including list boxes, pop-up lists, pop-up menus, spin-edit popups, and text-edit popups. An ADM list is composed of ADM entries. In similar fashion, an ADM Hierarchy List object is composed of ADM List Entry objects, which themselves may be hierarchical lists. See the following figure.




ADM hierarchy lists and list entries do not have many standard properties, like plug-in and bounds; rather, these are defined using the ADM hierarchy list's item. To access them, use `sADMListEntry->GetItem` to get the item owning the hierarchy list, then use the ADM Item suite functions (see "Item suite" on page 88") with the returned-item reference.

ADM lists and hierarchy lists have special properties, like a menu-resource ID and a group of entries. ADM entries and list entries have other properties, including an index and a selected state. These entry properties are used by the ADM List Entry suite to access the entries. The index is the position of the entry in the list. The selected state indicates the user has selected the item (others may be selected in the case of a multi-select list).

## Help support

**CAUTION:** The Help API is deprecated. This does *not* include tool tips.

ADM has built-in support for ASHelp, a WinHelp-type help system. This allows ADM user interfaces to use WinHelp-compatible files to provide assistance to the user. ASHelp uses WinHelp file definitions in a cross-platform fashion. Each ADM object can have a help ID that is used to identify a location in the help file to be displayed when help is triggered. The system can operate in contextual fashion. For example, selecting Command ? in Mac OS or Alt + F1 in Windows lets you click an item and see its help resource. For plug-ins to support help files, there must be a Plugin Help location in the `PiPL` resource.

## For more information

For Illustrator, see ADMListEntrySuite in *Adobe Illustrator CS3 SDK API Reference*.

For other products, see the documentation in the `ADMListEntry.h` header file.

# Notifier suite

The ADM Notifier suite lets you access the high-level events happening within your plug-in/user interaction. For low-level events, use the ADM Tracker suite functions (see "Tracker suite" on page 101).

## Accessing the suite

Acquire this suite by calling `SPBasicSuite::AcquireSuite()`. To determine the values to use for the name and version parameters in this call:

- For Illustrator, see ADMNotifierSuite in *Adobe Illustrator CS3 SDK API Reference*.
- For other products, see the `ADMNotifier.h` header file.

## Notifier functions

ADM notifiers are callback functions assigned to ADM objects. They allow your plug-in to be notified that the user has interacted with an object. A notifier function is be called when the user interaction is complete; for instance, when the mouse button is released. To specify an ADM notifier function to use with an ADM object, use an assignment function:

```
void ASAPI (*SetNotifyProc)(ADMObjectRef inObject,
       ADMObjectNotifyProc inNotifyProc);
```

For ADM Dialog objects and ADM Item objects, this assignment function is found in the object suite. Notifier functions for ADM Entry objects and ADM List Entry objects are assigned to the list or hierarchy list, respectively, that contains them. All entries or list entries in an ADM List object or ADM Hierarchy List object have the same notifier function.

All ADM notifier callbacks have the following signature:

```
typedef void ASAPI (*ADMObjectNotifyProc)(ADMObjectRef inObject,
   ADMNotifierRef inNotifier);
```

The *object* argument is a reference to the dialog, item, or entry that is to be notified a user event has occurred. The inNotifier argument is a reference to the notification event and is used with the functions in this suite to obtain information about the event.

All ADM objects have a default notifier function that provides their normal notification behavior. For instance, the default notifier for a kADMTextRadioButtonType is to set its selected state to true and the selected state of other buttons in its group to false. You always should call the default notifier function of an object to ensure standard behaviors occur. To call the default notifier, use a function of the object suite:

```
void ASAPI (*DefaultNotify)(ADMObjectRef inobject, ADMNotifyRef inNotifier);
```

Pass the DefaultNotify function the arguments that were passed to your notifier function; for instance:

```
void mySquareNotifyHandler(ADMItemRef item, ADMNotifierRef notifier)
{
    sADMItem->DefaultNotify(item, notifier);
}
```

## Using notifier functions

The functions in the ADM Notifier suite require an ADMNotifierRef, which basically is an event context. One of the arguments passed to your notifier function is a notifier reference, which is passed to each ADM notifier function:

```
void myDialogNotifyHandler(ADMDialogRef inDialog, ADMNotifierRef inNotifier)
{
    sADMItem->DefaultNotify(inDialog, inNotifier);

    if ( sADMNotifier->IsNotifierType(inNotifier, kADMZoomHitNotifier)
    {
        // handle the window zoom...
    }
}
```

## Notifier types

There are several types of ADM notifiers received at certain times, listed in the following table.

| Notifier | Purpose |
|---|---|
| kADMBoundsChangedNotifier | Received when an object is resized. This is received by both ADM items and dialogs. If a dialog receives this notification and resizes its items, the resized items would then receive this notifier. |
| kADMCloseHitNotifier | Received by ADM dialogs when a window's close button is clicked. It is your responsibility to hide the window; ADM does not do so automatically. |
| kADMCollapseNotifier | Received by ADM dialogs when the user is collapsing the panel via the tab. |
| kADMContextMenuChangedNotifier | Received by ADM dialogs when an edit operation occurs via a clipboard operation. |

| Notifier | Purpose |
|---|---|
| kADMCycleNotifier | Received by ADM dialogs when a user is double clicking or triple clicking in the title bar of a tab panel. |
| kADMEntryTextChangedNotifier | Received by ADM Entries when a list entry's text changes by in-place editing. For instance, when an edit-text item's text changes, a kADMUserChangedNotifier notifier is received. |
| kADMExpandNotifier | Received by ADM dialogs when the user is expanding the panel via the tab. |
| kADMGroupHideNotifier | Received by an ADM item group when it is hidden. |
| kADMGroupShowNotifier | Received by an ADM item group when it is shown. |
| kADMIntermediateChangedNotifier | Received by ADM items when a user is in the process of changing input data via a slider, etc., but has not yet completed the task. |
| kADMNumberOutOfBoundsNotifier | Received by ADM dialogs when a user enters a value greater or smaller than the min/max values for the entry. |
| kADMUserChangedNotifier | The default notifier, received by all ADM objects. This type applies to all notification events that cannot be classified explicitly as one of the precise types below. |
| kADMWindowActivateNotifier | Received by ADM dialogs when a window is activated (focus moves into the dialog area). |
| kADMWindowDeactivateNotifier | Received by ADM dialogs when a window is deactivated (focus moves to another screen area). |
| kADMWindowDragMovedNotifier | Received by ADM dialogs when the user moves the dialog by dragging it. |
| kADMWindowHideNotifier | Received by ADM dialogs when the user is in the process of hiding a window. |
| kADMWindowShowNotifier | Received by ADM dialogs when the user is in process of showing a window. |
| kADMZoomHitNotifier | Received by ADM dialogs when a window's zoom button is clicked. It is your responsibility to change the window size; ADM cannot do this automatically. |
| **Text-item notifiers:** | |
| kADMPostClipboardClearNotifier | Received by ADM text-edit items after the clipboard clear operation occurs. |
| kADMPostClipboardCopyNotifier | Received by ADM text-edit items after the clipboard copy operation occurs. |
| kADMPostClipboardCutNotifier | Received by ADM text-edit items after the clipboard cut operation occurs. |

| Notifier | Purpose |
|---|---|
| kADMPostClipboardPasteNotifier | Received by ADM text-edit items after the clipboard paste operation occurs. |
| kADMPostClipboardRedoNotifier | Received by ADM text-edit items after a redo command occurs. |
| kADMPostClipboardUndoNotifier | Received by ADM text-edit items after an undo command occurs. |
| kADMPreClipboardClearNotifier | Received by ADM text-edit items when a user issues a command to clear the clipboard, but the clear has not yet occurred. |
| kADMPreClipboardCopyNotifier | Received by ADM text-edit items when a user issues a copy command, but the copy has not yet occurred. |
| kADMPreClipboardCutNotifier | Received by ADM text-edit items when a user issues a cut command, but the cut has not yet occurred. |
| kADMPreClipboardPasteNotifier | Received by ADM text-edit items when a user issues a paste command, but the paste has not yet occurred. |
| kADMPreClipboardRedoNotifier | Received by ADM text-edit items when the user issues a redo command, but the redo has not yet occurred. |
| kADMPreClipboardUndoNotifier | Received by ADM text-edit items when the user issues an undo command, but the undo has not yet occurred. |
| kADMPreTextSelectionChangedNotifier | Received by ADM text-edit items when the user issues a text-selection change command, but the change has not yet occurred. |
| kADMTextSelectionChangedNotifier | Received by ADM text-edit items after the text-selection change operation occurs. |

ADM items automatically handle certain behaviors internally and not through their notification function. These behaviors include setting text values or pop-up list selections. If you want ADM items to interact, you need to use a notifier. As a matter of practice, you should always call the `sADMItem->DefaultNotify` function within your custom function, even though in many cases there is no default notification. The default behaviors of item notifiers are shown in the following table.

| Item type | Standard notification behavior |
|---|---|
| kADMFrameType<br>kADMListBoxType<br>kADMPictureCheckBoxType<br>kADMPicturePushButtonType<br>kADMPictureStaticType<br>kADMScrollbarType<br>kADMSliderType<br>kADMSpinEditType<br>kADMTextCheckBoxType<br>kADMTextEditMultilineType<br>kADMTextEditType<br>kADMTextPushButtonType<br>kADMTextStaticMultilineType<br>kADMTextStaticType<br>kADMUserType | None |
| kADMPictureRadioButtonType<br>kADMTextRadioButtonType | Sets the state of other radio buttons in the group. |
| kADMHierarchyListBoxType<br>kADMPopupListType<br>kADMPopupMenuType<br>kADMScrollingPopupListType<br>kADMSpinEditPopupType<br>kADMSpinEditScrollingPopupType<br>kADMTextEditPopupType<br>kADMTextEditScrollingPopupType | When the popup is used, notifies the item only if a list entry was selected. |
| kADMResizeType | Sends bounds-changed notification. |

## For more information

For Illustrator, see ADMNotifierSuite in *Adobe Illustrator CS3 SDK API Reference*.

For other products, see the documentation in the `ADMNotifier.h` header file.

# Tracker suite

The ADM Tracker suite lets you access the low-level events or actions happening within your plug-in/user interaction. For high-level events, use the ADM Notifier suite functions. The ADM tracker stores all the "tracked" state information in the TrackerRef, and it is a snapshot of what activity was in progress when the tracker was activated. Any GetTrackerInfo function obtains the state of the events at that point, not the current state.

## Accessing the suite

Acquire this suite by calling `SPBasicSuite::AcquireSuite()`. To determine the values to use for the name and version parameters in this call:

- For Illustrator, see ADMTrackerSuite in *Adobe Illustrator CS3 SDK API Reference*.
- For other products, see the `ADMTracker.h` header file.

## Trackers

ADM trackers are routines that track low-level user interaction with dialogs and dialog items. When used in conjunction with action masks (see `ADMTracker.h` for a complete listing of all trackable modifier keys and mouse-key actions), you can use trackers to keep aware of what the user is doing in interacting with your plug-in dialog.

For example, by setting up an ADM action mask with the various conditions you want to track and having your initialization routine set up a tracker procedure to call a notify procedure, your plug-in code can be alerted when the user causes an event you want to track.

The sequence is as follows:

1. Select the item to be "tracked," with `sADMDialog->GetItem`.
2. Set up a notifier routine to call when your user-interface button is pressed, with `sADMItem->SetTrackProc`.
3. Tell ADM what conditions to check for, with `sADMItem->SetMask`. When the specified condition is encountered, your Tracker proc should handle the appropriate response.

## For more information

For Illustrator, see ADMTrackerSuite in *Adobe Illustrator CS3 SDK API Reference*.

For other products, see the documentation in the `ADMTracker.h` header file.

# 5 ADM Folders and Files

The ADM files are part of an accompanying SDK for an Adobe host application. They provide the supporting files you need to work with your host application.

The following table lists the core ADM files.

| File | Description |
|---|---|
| `ADM.txt` | Definitions of ADM item types and Windows and Mac OS resource information. This is the same information listed in the tables in "Windows ADM items" on page 34 and "Mac OS ADM items" on page 38. |
| `ADMBasic.h` | ADM Basic Suite functions. |
| `ADMDialog.h` | ADM Dialog Suite functions. |
| `ADMDialogGroup.h` | ADM DialogGroup Suite functions. |
| `ADMDrawer.h` | ADM Drawer Suite functions. |
| `ADMEntry.h` | ADM Entry Suite functions. |
| `ADMHierarchyList.h` | ADM Hierarchy List Suite functions. |
| `ADMIcon.h` | ADM Icon Suite functions. |
| `ADMImage.h` | ADM Image Suite functions. |
| `ADMItem` | ADM Item Suite functions. |
| `ADMList.h` | ADM List Suite functions. |
| `ADMListEntry.h` | ADM List Entry Suite functions. |
| `ADMNotifier.h` | ADM Notifier Suite functions. |
| `ADMResource.h` | ADM Resource ID lists. |
| `ADMTracker.h` | ADM Tracker Suite functions. |
| `ADMTypes.h` | Definitions of ADM resources, units, fonts, colors, etc. |

# 6 ADM Error Codes

The following error codes can be returned wherever you see an `ASErr` / `ADMErr` returned by an API, or as the value of the `outError` argument to the `sADMBasic->GetLastADMError` call. In addition, you can return any of these errors from your initialization callbacks.

| | |
|---|---|
| `kNoErr` | `0` |
| `kOutOfMemoryErr` | `'!MEM'` |
| `kBadParameterErr` | `'PARM'` |
| `kNotImplementedErr` | `'!IMP'` |
| `kCantHappenErr` | `'CANT'` |
| `kADMCustomResourceError` | `'rErr'` |
| `kADMCustomResourceExistsError` | `'!Unq'` |
| `kADMStreamUnavailableError` | `'noSe'` |
| `kADMResourceNotFoundError` | `'r!fd'` |
| `kDialogResourceNotFoundError` | `'DLOG'` |
| `kDialogItemListResourceNotFoundError` | `'DITL'` |
| `kCouldntCreateItemError` | `'!itm'` |
| `kDockHostConflictError` | `'DOCK'` |
| `kTabGroupNotFoundError` | `'T!FD'` |
| `kAlreadyDockedError` | `'DCKD'` |

For more information on errors returned by Illustrator, see Error Return Codes in *Adobe Illustrator CS3 SDK API Reference*.

# 7 Frequently Asked Questions

## Lists

**My hierarchical list entries are not showing. What do I need to do?**

Call `sADMHierarchyList->SetEntryTextRect`. The text rectangle for hierarchical lists defaults to an undefined rect.

**I have an ADM frame item in my dialog with items inside. Those items do not seem to work. How do I make them work?**

Move the frame item lower in the item list, so it is the last item created. Do not just change the ID; open the `.rc` file as text, and move the line defining the item.

**I'm trying to get a scrolling list box that allows items to be checked and unchecked. Is there an easy way to do this in ADM?**

There is no built in way, but it is not hard to implement. Essentially, make two icons, checked and unchecked, and put a track proc on the list. When the mouse clicks the icon, use the `sADMItem->SetPictureID` function to change the icon. Then call the default.

**I've seen a reference to kADMNewDeleteStyle, but it seems to be gone. Is there an equivalent? (I want a trash can at the bottom of my list, and I'll be adding a few custom icons too.)**

Add the items yourself as pict push buttons. You can use the resize item for a relative position and height. The icons are included in ADM, and you can set the picture ID to one of ADM's constants. See `ADMResource.h`.

**My list flickers when it draws. Is there a way to reduce this?**

There is support for drawing entries in a list box off-screen. Just add the `kADMUseImageListBoxStyle` style.

**Is it possible to hide the scrollbar in an ADM ListBox? I have a list box that will have a fixed number of items and does not need the disabled scrollbar.**

Use the following code to hide the scrollbar:

```
scrollbarItem = sADMItem-
>GetChildItem(listItem,kADMListBoxScrollbarChildID);
sADMItem->Show(scrollbarItem, false);
```

# Text

**I have an ADM numeric text item that has a min and max, but it can also have no value to indicate it is unused. When the user deletes the value, ADM puts a 0 back in the field. How can I get it to stop this behavior?**

Use an ADM TextToFloat function (see "FloatToText and TextToFloat functions" on page 92). ADM has a default TextToFloat routine it uses. You can override it by using `sADMItem->SetTextToFloatProc`, and you will have an opportunity to affect the float value used for the item.

If the TextToFloat proc returns `false`, the text is assumed to be invalid, and a notification is presented to the user. If `true` is returned and the item is known (see `sADMItem->IsKnown`), ADM checks it against the min and max values and acts accordingly.

If `true` is returned and your TextToFloat proc marked the value as unknown, the text is used as is, and no notification to the user is presented.

**I want to have a static text label select the text edit item next to it. How do I do this?**

`kADMTextStaticType` and `kADMPictureStaticType` items auto-activate the edit items they are next to. By default, they look at the next item to see if it is editable. If not, they try the previous item. If that still is not editable, they try the one after the next item. (The last case is to handle labels on a slider that have edit boxes.) This works in most cases.

if this is not the behavior you want, you can use `kADMDisableAutoActiveTextStaticStyle` and `kADMDisableAutoActivatePictureStaticStyle` to turn off auto-activate.

If it does not seem to work, check the ordering in the resource. Alternately, disable auto-activate and, on the `kADMUserChangedNotifier`, activate the correct edit item.

For Windows static-text items, `kADMDisableAutoActivateTextStaticStyle` also disables converting ampersands to underscores for shortcuts in modal dialogs. Any Windows static-text items that display user defined strings should turn this flag on in their dialog init code.

**How do I tell if a text item is part of a composite item?**

If you use composite items, you would need to keep track of the parent item and check for matches against its children using `GetChildItem`.

**Why does every edit item display with "pt" appended to the text?**

This is the default. You can turn off units for the individual item with the `sADMItem->SetUnits` function. You can set the defaults for all new text items with the host suite `SetADMDefaultTextInfo` function.

**I can't get an edit text to display centered. Why?**

This would be on Windows, since the `sADMItem->SetJustify` function works in Mac OS. Windows does not support changes in justification to a single-line edit item after it is created.

**Is there any way to measure how many pixels wide a particular string displayed in a text item will be?**

No. Only static text can be measured.

**How do I get ADM numeric text fields to not allow decimal numbers. When I set a precision of 0, I can still enter, for instance, "5.5," and it will truncate to "5.0."**

Set `kADMIntegerNumeric` instead of `0`.

**There is no mechanism for appending text to the text edit item. I can get the text, append to that, and re-set the text, but that's not an efficient solution for an interactive console, especially once the console contains a lot of text.**

Use `sADMItem->GetChildItem`. The child IDs are listed in `ADMItem.h`.

**Is it possible to access the scroll bar for this multi-line text item (and if so, how do I get a reference to it)?**

No.

**I'd like to change text fields back and forth between edit-text and static-text based on the settings of check boxes. This works in the Mac OS dialog manager. I can't just disable an edit-text item, since text in disabled edit-text boxes disappears. That's annoying, as I'd like users to see the text, just not edit it.**

You can make an edit and a static text. Instead of doing a set type, flip the one that is visible.

**I'd like a call in the ADM Entry suite to change the text style.**

You can override the drawer for the list, and since you would use the ADM Drawer suite, it will be cross-platform.

**I want to set the text on my list item. Is the basic idea as follows (using IADM)?**

```
IADMEntry e = hl->InsertEntry(1);
IADMItem i = e->GetItem();
i->SetText("foo");
```

You would not set the text on the item, but on the entry itself. You probably do not need the item reference/object.

```
IADMEntry e = hl->InsertEntry(1);
e->SetText("foo");
```

**What are the string lengths in use by ADM?**

For the most part, and for ADM object text, the length is 256 (including the terminator). Tool-tip text is allocated dynamically, since tool tips may not be in use. Text identifiers (e.g., dialog names) are pooled strings. As such, they are subject to the SuitePea string pool limits. By default, there is no maximum length, but SuitePea allows the string pool to be substituted and, so, it is subject to the host.

**My edit control is defaulting to a numeric style control and ADM tries to validate it and gives an error if there was ASCII text in the control. The resource definition looks like the following. How can I get it to be a normal text item by default?**

```
EDITTEXT ctUser1,102,35,70,12,ES_AUTOHSCROLL | ES_OEMCONVERT
```

Add the `ES_LEFT` style as:

```
EDITTEXT ctUser1,102,35,70,12,ES_AUTOHSCROLL | ES_OEMCONVERT | ES_LEFT
```

# Color

**I want to fill a rect with color. How do I do this?**

Use an ADM user item and override the Drawer proc. In the your Drawer proc, you can use `sADMDrawer->SetRGBColor` and `sADMDrawer->FillRect`.

**Tell me where I'd get the proper ADM calls for tool bevel colors (3DShadow, 3D highlight, 3D Fill, etc.)?**

The ADM colors are in `ADMTypes.h`. Some of the colors may have the same RGB values, but by using the constants, the user interface can change and will look correct.

For a given ADM drawing operation, you can set them with `sADMDrawer->SetADMColor`. You can get the RGB representation of an ADM color with `sADMBasic->ADMColorToRGBColor`.

**How is kADMForegroundColor used? Is there a foreground color that is stored even when you've switched to something like kADMBackgroundColor? If so, what is the proper way to save/restore the color state? Does calling `sADMDrawer->SetRGHColor` implicitly switch you to kADMForegroundColor mode?**

Image-processing programs likely have a concept of a foreground color and a background color that affect certain operations. ADM's color scheme is very simple, being intended for user-interface work. The foreground color is always black (or gray for disabled items). That's it—no modes, just black. The background color is some variant of grey.

To get and restore colors, use the ADM Drawer color functions:

```
sADMDrawer->GetRGBColor
sADMDrawer->SetRGBColor
sADMDrawer->GetADMColor
sADMDrawer->SetADMColor
```

`sADMDrawer->GetADMColor` returns either the user-interface color constant in use or the RGB color, so it may be more convenient in some cases since you would not need to do a look-up yourself.

**Should I save/restore both the RGB color and the ADM color? Or is the state reset to default at each draw event? Is my drawer private to me?**

A drawer is created for each drawing operation, and whatever defaults exist are used. So, in general, your drawer is reset and ready to use. There is one known bug with the clip rect for ADM entries, so we cannot say it always is fully reset. Assume it is, and report strange behavior to Adobe Developer Support.

**How does ADM handle the window's system colors?**

ADM recolors bitmaps in `.icn` format, but not those in `.bmp`. The recoloring scheme recolors all shades of gray in the original image. Colors are not recolored, and you can prevent recoloring by making pixels slightly off-gray.

If the face color is dark, it maps a dark shade of gray; if it is light, it maps a light shade of gray. Everything else maps proportionally. So, for example, if the color scheme has the face at 25% intensity, the highlight at 10%, and the shadow at 50%, the grays map accordingly (50% gray becomes the shadow color when mapped). If the scheme is darker and has the face at 50%, the highlight at 25%, and the shadow at 75%, the grays still map accordingly (50% gray becomes the face color when mapped).

# Panels

### How do I set and restore how panels are tabbed together?

For sample code, see the ADMNonModalDialog sample on the Adobe Illustrator SDK. The gist of the process is described below.

The following two functions are involved: `sADMDialogGroup->GetDialogGroupInfo` and `sADMDialogGroup->SetDialogGroupInfo`.

The Get function is used to retrieve the current panel positioning information, so you can save it in your preferences. The Set function is used to restore your panel position. You need to have some default information for the first time the panel used.

Use the Get function in your dialog Destroy proc. This way, whenever your dialog is destroyed its position is saved; you do not have to worry about it being saved at shut-down or whenever else saving might need to occur. Similarly, use the Set function from within your dialog Init proc.

There are two values needed to save a tab panel position: `groupName` and `positionCode`. When you use the `sADMDialog->Create` function, you specify a dialog name. This is meant to be a unique identifier and is used to determine a panel's position. The `groupName` returns this identifier for the top-left-most panel in a tab/dock group. All panels in a group should have the same `groupName` identifier. The front tab is dealt with elsewhere; the `groupName` panel is an anchor.

The position code identifies where a panel is relative to the top-left panel (`groupName`). There is information on this in the `ADMDialogGroup.h` header file; in a nutshell, it is a four-byte code. The low two bytes are 1-based indices specifying where the panel is relative to the top left one. `0x00nn0101` is the top left, `0x00nn0201` is the second tab in the top tab group (top dock), `0x00nn0102` is the first tab in the second tab group (second dock), `0x00nn0202` is the second tab in the second dock, etc.

The third byte is a group of bit flags. (There are masks in `ADMDialogGroup.h`.) Bit0 indicates whether the current dialog is the front-most tab; only one per tab group would have this set. Bit1 indicates whether the tab group is collapsed (`1` is collapsed, `0` is expanded); all panels in a tab group probably would have the same setting. Bit2 indicates whether the entire dock group (all tabs and all tab groups) is visible; all panels in the collection would have the same setting (if not, the last one to set a position code would set the visibility for all).

In your preferences file, just write/read a C string and a `long`.

In Illustrator, there is a file with all the panel name constants and dock codes. Since they are all relative to each other, they should be in the same place.

### Do I always have to create all my panels?

A panel would not have to be created if it is not visible and standalone (not tabbed or docked with anything else; there is a function to determine this for a given position code). Other times, a panel must be created, because it is in a group or visible but standalone. The tendency is to create all panels all the time.

### Does the order in which I restore my panels matter? Do I have to restore a group completely?

No. During restoration, the top-left dialog does not have to exist initially or at any time in the process. Other tab panels can be missing and it will not affect the process. So, for instance, if you are operating in a plug-in panel world, panel restoration works even if one or more plug-ins is removed.

**How do I create the pop-up list for a panel—the one in the upper right of the window by the tabs?**

Get the `kADMMenuItemID` item and, from it, get its list. When you populate the list with either a platform menu resource or by creating entries, it automatically is made visible.

**How do I do a dialog tracker?**

Most panels/dialogs that need tracking need it at the item level. If they need to track the entire window, probably there is one or more items that cover that window. Simply make a tracker as big as the window or, for instance, as big as the window less scroll bars, and assign a tracker to it. Dialog trackers can be tricky to use.

Dialog trackers are different than item trackers; a dialog track proc should not expect that the actions sent to it are similar to those sent to an item's track proc. Dialog trackers can be used when the items in the dialog do not want to handle an event. The ADM tracker sends that event to the item's dialog, to see if it can process the event. You can set up a dialog tracker to process such an event.

For example, a use of a dialog's track proc would be to trap keyboard events/modifiers for making shortcuts work.

**We are laying out our ADM panels and dialogs with PICTs for picture buttons. The backgrounds of the picture buttons sometimes differ from that of the dialog. How does ADM handle the system colors?**

Do you mean, how do you get your pictures to blend in with whatever color scheme is being used? With PICT/BMP resources, you cannot; they always retain their panel. The preferred way of doing picture buttons is using icons, because of their masking feature, enabling backgrounds and other system colors to show through.

**How can I tell when my panel window is visible?**

There are two notifiers for this in `ADMNotifier.h`. Dialogs should receive hide and show notification. The simplest case for this is when the close box is hit.

When a dialog is tabbed with others but not front-most, it is considered hidden. When it is brought to the front, the show notification is received. When another tab is selected, the hide notification is received.

All dialogs that are part of a dock group and the front-most tab of their tab group receive hide/show notification based on a hide/show action to any other front tab in the dock group. This is because a hide/show on any visible panel affects any docked panels similarly.

Using the Tab key to hide/show all panels triggers this notification for any visible dialog.

The collapsed state of a tab does not affect its visible state and, thus, its corresponding menu state. Collapsing and expanding a tab does not trigger a hide/show notification.

Also when saving state information to the preferences file, the visible state of tabbed panels should not be saved directly; the visibility of all panels docked together is noted in the group position code.

**Is there a desirable width for floating panels?**

The default Adobe panel width is 206 pixels.

**The multi-line text-edit field includes a scroll bar. Normally this is useful, except when I size the field to the full extent of the dialog. In this case, the down arrow on the scroll bar is hidden by the close box. I normally would fix this by making the scroll bar shorter than the text field itself.**

All Adobe panels are like this, presenting information above the resize box and using the area to the left for things like buttons. Your design should follow this layout and provide shortcut buttons (or pictures or white space).

**My panel windows lose focus when the return key is pressed. This is interfering with my track proc's ability to get key events.**

The window with focus is the one that receives key events. Even though an ADM window is front-most, it does not necessarily have focus. This is by design, so the document window has priority. The user can assign a panel window to have focus by doing a Cmd/Opt or Ctrl/Alt-click on the window. It should then receive all key events.

**I just started using ADM to implement the tool panel and am having problems getting the pop-right flyout tools menu to track. I am creating a kADMPopupDialogStyle window and then adding kADMPictureRadioButtonType items to it. I receive the mouse-down and -up messages to show and hide the submenu through the tracking proc fine, but I don't get any mouse-movement messages to track the menu. Should this work, or do I have to track the menu via the Notify proc using the kADMIntermediateChangedNotifier message?**

The ADM item in which the mouse down occurred captured the mouse, and all mouse actions go to that item until the mouse is released. So you should be getting MouseMovedDown actions for the item in which the button press occurs, which you must translate into the space of the flyout to figure out what tool it is in.

The mouse-down item would "uncapture" the tracker, which then would allow events to be dispatched to the items in which they occur. The mouse-down item would get an additional UncapturedButtonUp event when the mouse button is released, so it could know to pop the tool menu down.

# Dialog-box Behavior

**I'm trying to create a resizable ADM dialog but am unable to figure out how to get items inside of the dialog resizing correctly. The things that I've tried all end up in a stack overflow. How do I do this?**

Attach a notifier proc to the resize item instead of the dialog. The notifier proc should look for bounds changed.

**I want to bring up a system modal dialog and need to de-activate the ADM windows. How is this done?**

To support de-activating panels in Mac OS when you bring up a system modal dialog, there is the ActivateWindows call in `ADMHost.h`.

**Why does a dialog have a name, an ID, and text?**

The ID maps to the resource ID. The text is displayed in the tab or the title bar of the window. The name is an internal identifier and is not displayed to the user. It is used to restore dock groups and may be used for identifying resources in the future.

**Calling `sADMDialog->SetText` for a dialog of type kADMFloatingDialogStyle doesn't make the title appear. It seems as if your WDEF doesn't display them (I am working on Mac OS). Is this by design, and is there any way around it?**

`sADMDialog->SetText` is the correct way to do this. The text appears in the tab for tab panels. Adobe does not put text in our title bars, so for non-tabbed windows, it was not hooked up.

**I am experiencing problems getting ADM to load resources that are attached to the application and not a plug-in. Is there a way to set where ADM looks for resources?**

When you add a dialog or panel (e.g., `sADMDialog->Create`), you specify a plug-in ref, even for the application-base ones. What did you specify here? You should create and save a single host plug-in (`SPAddHostPlugin`) and use this for all your application-defined dialogs. When you add the host dialog, you can specify a file instance to use for resources. ADM will use this.

**How do I draw into an ADM window at an arbitrary time?**

You can invalidate the item, and ADM calls your Drawer proc during the next update. If you need to do something like drag feedback where this is not possible, you can create a drawer for the window port. Remember to release it, though.

```
ASWindowRef windowRef = sADMDialog->GetWindowRef(sADMItem-
>GetDialog(item));
ASPortRef portRef = sADMDrawer->GetADMWindowPort(windowRef);
ADMDrawerRef drawer = sADMDrawer->Create(portRef, &boundsRect,
kADMDialogFont);
DrawFeedbackXOR(drawer, location);
sADMDrawer->Destroy((ADMDrawerRef)drawer);
sADMDrawer->ReleaseADMWindowPort(portRef);
```

**Why don't my dialogs remember their last location on the screen, and instead always pop up in the same place?**

The "same place" should be centered on the screen, which is the default location if you do not bother putting them somewhere else. The ones that remember their location are ADM dialogs where the plug-in does a `sADMDialog->GetBoundsRect` in the Destroy proc and `sADMDialog->SetBoundsRect` in the Init proc. Some of your dialogs may be system dialogs (e.g., print/file-related ones); there is nothing ADM can do for you there.

**`sADMDialog->InvalidateRect` causes an update of the whole dialog. Is there a workaround for this?**

Invalidate the individual items and entries.

**Is it possible to change the type of an item after a dialog is created?**

You cannot change dialog or item types after they have been created.

**The modal dialog I create doesn't prevent Photoshop from switching out. I'm using `sADMDialog->Create` with kADMModalDialogStyle, and then using `sADMDialog->DisplayAsModcel`, rather than using `sADMDialog->Modal` directly. Is that OK?**

In the Adobe user interface, you are supposed to be able to switch out of the application.

**What is the correct way to delete a dialog? In my dialog Notify proc, I look for the kADMCloseHitNotifier notifier. I then call `sADMDialog->DefaultNotify` then `sADMDialog->Destroy`, but ADM crashes.**

The way this is implemented, you are essentially deleting it while it is still in use. The crash is expected. Note the dialog that needs to be deleted, and call `sADMDialog->Destroy` during your idle proc.

**I want to dismiss a modal dialog with a shortcut. How does *`sADMDialog->EndModel`* work?**

The `sADMDialog->EndModal` function takes a parameter, `ASBoolean inCancelling`. Normally this should be `false`, but it can be `true` if you are implementing another way of getting the Cancel functionality. When you call `sADMDialog->EndModal` with `inCancelling` set to `true`, ADM does not verify the text in the current selection. If `inCancelling` is set to `false`, `sADMDialog->EndModal` might return `false` (this is the place where you probably have to change your code); if it does, your code should stop processing the notification and act as if nothing happened. What is happening in this case is that a numeric text item had the focus, the user typed in something illegal, ADM put up an alert, and so your code should cancel the notification handling.

**I have a dialog box that uses `sADMItem->SetFont`, passing in the kADMPaletteFont on some items and normal on the other text items. Everything is fine on the Roman side: small text is small, and normal text is normal. But when I run the US Photoshop on a Japanese operating system on Windows, the text is all the same, normal size. My dialog box is very busy, and I need the small text to be small. How do I get around this?**

You cannot. Non-Roman fonts tend to be less usable at smaller sizes, so though they exist, they are not supported.

**How should I handle an error in my Init proc during modal-dialog initialization?**

Do not use `sADMDialog->EndModal` in your initialization procedure. Instead, when an error occurs during initialization, the Init proc should return something other than `kNoErr`, and the dialog will not start up or show up.

# Popups

**I want to have a pop-up menu at the top of my preferences dialog, so I may have multiple sections. (Basically the same kind of functionality as the standard Adobe preferences dialog.) What is the solution with ADM?**

All the items exist within one dialog and are hidden/shown or moved on/off-screen as needed.

**In response to certain user actions (for instance, a double click), I pop a modal dialog from within a tracker. I get some odd behaviors on the trackers I add to the new dialog.**

The code used to display the dialog resides within the Tracker proc, resulting in nested trackers. ADM is not robust when it comes to handling nested trackers.

The thing to do is note that a double click occurred (and any other relevant info), and then call `sADMTracker->Abort` and return `true`. This allows ADM to clean up the current tracker and causes the notifier proc to be executed. In your notifier proc, check for a user-changed notification and your double-click flag, then pop the dialog. All should be well. If you can get the platform port for an AGM port, you can create an ADM drawer for it, although AGM support in ADM is deprecated.

**How can I set a pop-up item list to use the small (panel) font?**

You cannot mix fonts in dialogs (with the occasional exception of static text).

**There's no call in the ADM List suite for selecting a single item.**

You could just get the active one and de-activate it, followed by activating the new active one.

# Dialog-box Elements

**What is an intermediate change notifier?**

When a user action is completed for an item, like a button press or tabbing out of a text item, a `kADMUserChangedNotification` is sent to the item. For sliders, where the user clicks on them and moves them, you might want some notification before the user lets up on the mouse. In this case, ADM sends `kADMIntermediateChangedNotifications` to the item. When the mouse is released, the user-changed notification is sent.

**How can we display a string after our name in the Splash Screen, so we can put up something like "Initializing QuickDraw 3D"?**

It goes something like the example below. This works only during the start-up message. The string you pass should appear in the splash screen. Also, see `SPRuntme.h`.

```
SPErr error = kSPNoError;
SPHostProcs *gHostProcs;
error = sSPRuntime->GetRuntimeHostProcs(&gHostProcs);
if (!error)
{
gHostProcs->startupNotify(kSetMessage,
(void*)"Initializing QuickDraw 3d", gHostProcs->hostData);
}
```

**When I create a slider, it won't do anything.**

Make sure you set the min and max. They are the same by default.

**Resizing the field (by resizing the window) causes the scrollbar to reset to zero and changes the selection.**

Get a resize notifier and set the position wherever you think it should be.

**How do you recommend doing icon toggle buttons (i.e., an icon button that toggles on and off, rather than one that's off by default and turns on only when you hold the button down on it)?**

There are picture check buttons, `kADMPictureCheckBoxType`. This should do what you want. There also are picture radio buttons for use in a group.

**What are the resource parameters for creating a kADMPictureCheckBoxType?**

See the platform-specific resource information in the tables in "Windows ADM items" on page 34 and "Mac OS ADM items" on page 38.

# Timers

**I start doing something in my intermediate notification, and it takes a long time and the control isn't very responsive anymore. What do I do?**

You either scale back on what that "something" is, or you set a timer to see if the user has paused for a duration and do the update. ADM timers work especially well in this case.

**How do timers work?**

A timer takes an item or dialog reference, a duration in milliseconds, and two callback procs. The first callback proc is called if the duration expires; from the completion proc, you can return `true`, and ADM repeats the timer. The second callback is called if the timer is aborted before the time duration. In this case, an abort mask is passed to the create timer call (see, for example, `sADMItem->CreateTimer`). If one of the actions in the mask occurs before the timer duration is finished, the abort proc is called. The action that caused the abort is passed to the callback.

# Operating-system Related Queries

**How does ADM handle platform native resources?**

ADM uses platform native resources edited with the platform resource editor, so you have to maintain two sets of resources, for Mac OS and for Windows.

**Can I use a Windows ActiveX control within an ADM item?**

Yes, it has been done. This is not explicitly supported by Adobe Developer Support, so you are on your own in doing so. For possible help, see the Q&A for MFC controls.

**Is ADM thread safe?**

ADM is as thread safe as the operating system on which it runs. This is because ADM items are implemented using platform controls.

**Can ADM panels/dialogs receive operating-system-level drag events?**

Yes, it is possible. It is a host- or platform-provided service, meaning there is no support for drag-and-drop built into ADM. For instance, Illustrator implements drag and drop between the main document and the panels and exports this as a SweetPea suite. This allows both the application and plug-ins to take advantage of it by acquiring the suite/functionality.

What essentially happens is the subscriber (application or plug-in with a panel) registers the data types it can accept and a callback for the drag action. The callback handles the feedback within the ADM dialog using standard ADM dialog, item, and drawer functions.

**The call `sADMItem->SetCursorID` is failing in Mac OS but works in WIndows.**

The Mac version is missing the cursor resources. ADM recognizes `'CURS'`, not `'crsr'`.

**When you move any of the panels in Windows, the panel outline border is the same color as the main window. This results in not being able to see the panel outline when moving it. Expected results: use a different border shade/color from the main application window to make it visible. The default Windows color scheme was used. Is there anything we can do about it?**

Not on the ADM side. This is a system setting. If you use the display-control panel and turn off "show window contents when dragging," all applications have this behavior. Live dragging was disabled, because docked panels actually are several windows.

**Can I have a notifier function set for an event as a Windows or Mac OS event?**

No. Notifiers are assignable to an ADM object, not an event. If you want to get low-level events, use a tracker. ADM provides a cross-platform event mechanism, and there is no way to get the system event.

**Severe flashing occurs with the insertion cursor and text as you add or delete words. In some cases, the issue is so bad that the insertion cursor fails to correctly line up at the location of where you really are editing.**

This is standard Windows behavior. Windows does not know how to set an insertion point in a single-line edit text.

**I can't get floating-point sliders to work with a range of [0,1]. I use the following for initializing, where min, max, incr, and value are all floats. Using min = 0, max = 1, incr = .01, and value = .5, the slider thumb pops back and forth between 0 and 1 (the left and right endpoints). (Workaround: I gave up and scaled the values into and out of the sliders.)**

```
sItem->SetMinFloatValue(i, min);
sItem->SetMaxFloatValue(i, max);
sItem->SetSmallIncrement(i, incr);
sItem->SetFloatValue(i, value);
```

In Mac OS, the slider is a custom `CDEF`. This means that internally the values are integers. On Windows, the slider code explicitly casts to an integer and then to a float, perhaps for compatibility with the Mac OS `CDEF`.

**`sADMDialog->Create` appears to be ignoring the "initially visible" bit in the Mac OS DLOG resource. Is there a way to make a dialog initially invisible, so I can set a bunch of parameters and move controls around without being visually disturbing?**

In your Init proc, you should do all your set-up. The default `Show` does not occur until after initialization, which should give you the effect you want. In your Init proc, you could do a `sADMDialog->Show` with the Boolean parameter set to `false`. The window will not appear until you do a `sADMDialog->Show` with the Boolean parameter set to `true`.

**Is there a way to set the style of items in pop-up menus? I need to sometimes make items bold and/or italic. Do pop-up menus use a Mac OS menu on Mac OS? If so, is there a way to get access to it?**

No.

**PICT items in Mac OS dialogs are stretched on display to fit the item rect, while in ADM, PICTs are apparently centered. Is there a way to get stretch-to-fit?**

No. Adobe interfaces are pixel-perfect, so we would use the correct size picture.

**It appears to me that Windows ADM doesn't attempt to control the font used in pop-up menus. On my machine, the popups appear to use MS Sans Serif instead of Adobe UI. The MS Sans Serif font does not have all the glyphs we have in the Adobe UI font, so when we try to use fancy quotes in a pop-up menu, they just show up as a vertical bar.**

ADM uses standard platform controls where it can, and this is one of those cases.

**On WIndows, Invalidate and InvalidateRect do not erase the background when repainting.**

Use the Windows invalidate calls with the `bErase` flag set to `true` to force the background to repaint.

**How do you specify ADMHierarchyList control in a DIALOG resource on Windows?**

Add a custom control item to the dialog layout when in the resource editor, then name the "class" of the custom control to be "ADM Hierarchy List Box Type" along with quotation marks.

# Other

**Can I mix platform functions such as the following with ADM suite functions?**

```
ShowWindow(admGetWindowRef(dlgRef1), SW_SHOW);
```

No. There often are additional housekeeping things ADM does that would likely get out of sync if the direct message is sent. Also, using the `sADMDialog->Show` function keeps your code cross-platform.

**I don't see anything in ADM to handle customized standard file dialogs (and their Win95 equivalent).**

ADM provides a standard file dialog, but the only customizable bit is the file filter proc that can be passed to it. There is no provision to add items, and since this is a system dialog, there is no simple way to append an ADM item to it in a cross-platform/product manner.

**I'm hard-coding a rectangle to tell me where to do a Photoshop::DisplayPixels. Can this be expressed in a resource?**

ADM provides a user item you can use. It will do nothing if it is not assigned drawer/tracker/notifier procs, so it can serve as a placeholder. The bounds can be retrieved with an ADM Item call.

**I need to be able to have ADM not adjust the cursor over an item, as we want to do our own cursor adjustment and not have ADM change it back.**

There is a cursor constant you can set, and ADM will not bother changing the cursor at all: `kADMHostControlsCursorID`.

**The following routine is ignoring the passed rectangle. Has this been fixed already? If so, can I assume that the invalRect is passed in item-local-coordinates?**

```
void ADMEdgeItem::InvalidateRect(IASRect &invalRect)
{
if (GetWindowRef())
::InvalidateRect(GetWindowRef(), nil, false);
}
```

The implementation of this in some versions of ADM did invalidate the whole item. In current releases, the rect is local.

**I want to handle the clipboard operation in a special way. How do I do this?**

You need to override clipboard operations within an ADM dialog. To override the clipboard operation in a dialog, for all edit-text items in the dialog, attach a notifier that watches for the `kADMPreClipboard*` notifiers below. When you detect one, you can inspect the edit state and decide whether to allow it. For instance, you might look at the selection range.

If you decide you want to handle it in some fashion, call the notifier suite function:

```
void ASAPI (*SkipNextClipboardOperation)(ADMNotifierRef notifier,
ASBoolean skip);
```

ADM will not do the default clipboard action. Presumably, you would do something appropriate within your notifier function. See the text-item notifiers in the table in "Notifier types" on page 98.

**Do you have any examples I could start from, such as an ADM reference application?**

There are examples of the use of ADM from within plug-ins in the SDKs for Photoshop and Illustrator.

**Is there any way to set up an ADM drawer to draw into an AGM off-screen port? I have a UI item that I draw with AGM and then copy to the screen. It now has an icon as part of it. What I'm doing now is copying the AGM port, then using ADM to draw the icon. But when the item changes, it flickers. Can I use ADM to draw the icon into the AGM port? I also am using AGM to draw into an off-screen port for later use. I need to draw some text into this using the UI font. Drawing text with AGM is difficult and if I could use ADM to do it, it would be much easier.**

AGM support in ADM is deprecated. You can perform off-screen drawing in a Drawer proc with the ADM Image suite:

```
void ASAPI myDrawProc(ADMItemRef item, ADMDrawerRef inDrawer)
{
    imageRef = sADMImage->CreateOffscreen(width, height);
    if(imageRef != nil)
    {
        offscreenDrawer = sADMImage->BeginADMDrawer(imageRef);
        // draw stuff with offscreenDrawer

        sADMImage->EndADMDrawer(imageRef);
        topLeftPoint.h = 0;
        topLeftPoint.v = 0;
        sADMDrawer->DrawADMImage(inDrawer, imageRef, &topLeftPoint);
    }
}
```

**I have written a plug-in, and now I am implementing (functionality implementation). What I am trying to do is to have a group of buttons, with each button responding in the same way. So I would place all buttons in a group box. Now, when implementing in my plug-in the ideal way would be to get *one* notification when any of the buttons is being pushed that is on the group. How would I implement this using ADM? Can I attach callback function directly to the group?**

Use an ADM item group.

**We want to have Cmd/Ctrl do something like channel selection in Photoshop. ADM provides the Illustrator behavior (select the last active dialog). How can we change this?**

ADM knows nothing about the Photoshop behavior, or what panels exist in the host application and how to activate them. It can do nothing for this custom behavior.

You have two options:

- Handle this on the host end by trapping this key code before calling `HandleADMMessage/HandleADMEvent`, to keep ADM from handling it. Then implement whatever behavior you want. Do not feed it to ADM.
- Accept the default ADM behavior, and let Photoshop differ.

**In my multi-line edit field, double clicking a word then dragging left/right/up/down does not highlight as you drag as it does in text-edit programs. Single click and drag works normally though. I can't get kerning and tracking to preview in my ADM multi-line text-edit item. Pressing the Up or Down arrow does not take you to end-of-line or beginning-of-line.**

ADM does not claim to offer a word-processing text item. Standard platform text tricks are supported.

**Sometimes when I click for an insertion point, it takes two or three seconds for the insertion cursor to catch up. This tends to happen after a lot of use of the type dialog or using large anti-aliased type.**

You probably have the notifier doing some hefty processing when the mouse is clicked. There are no interruptible ADM notifiers.

**I have some cases where the ADM CDEFs (for buttons, combo-boxes, etc.) aren't found. I think what's happening is that in the cases where plug-ins are not moved to the top of the resource chain, the CDEFs which are in the application aren't found because it is at the top of the chain.**

The expected resource chain order when ADM is used is application-ADM-plug-in. SP plug-ins usually get this for free. There are some host callback procs which, if specified, allow you to set the resource chain for a SPPluginRef before ADM tries to do a resource access.

**Am I right in assuming that if you cannot enter units, you can't enter math expressions?**

No, units and math are orthogonal. You can do math without units, but you cannot specify units in the operands.

**How does the "known" state for an ADM item work?**

An item is in a "known" state if it has a "good" or valid value. For example, setting a check-box item to `known(checkboxItem, false)` sets it to an intermediate state. The check-box item then becomes "known" when it is checked by the user. The only way for an item to become "unknown" is by using the `known(someItem, false)` API. As another example, if you set a text item to unknown, it clears itself.

If you set the value of an item through the Set interfaces, it becomes known. If the user enters a value in a text item, it becomes known. If the value of a text item is unknown, it reverts to being empty in error conditions instead of reverting to its current value. If you have an item that you may set to unknown, check `sADMItem->IsKnown` before getting its value.

Another example can be taken from the `kADMSpinEditPopupType` item. For the spin-edit as whole, "known" means the numerical value is known. Setting the parent item to unknown makes all its child items unknown, but setting the parent item to known does not make its pop-up child known. (If the

value is not on the menu, the popup can be unknown even though the spin edit is known.) If the popup is becoming known, it will do so by virtue of its value being set or by an entry being selected.

**What is the return value for a track proc?**

The return value from a tracker procedure can mean two different things. For keyboard actions, it indicates whether the tracker "ate" the keystroke or whether it should be propagated out to the surrounding environment. For other actions, it indicates whether to call the notify procedure. Some cases have arisen where the latter meaning is required for keyboard actions.

**Does ADM provide a way to get event information outside of a track proc?**

For the ADM Tracker suite, if you pass `NULL` for `sADMTracker->GetModifiers`, `sADMTracker->TestModifier`, and `sADMTracker->GetPoint`, they return the current state. (Normally they return the state of the keys when the event occurred in the tracker.) So, for example, if you need to check the state of the Opt/Alt key when your menu item is chosen, you would call the following:

```
if (sADMTracker->TestModifier(NULL, kADMModKeyDownModifier))
```

**I am trying to use `sADMHierarchyList->SetInitProc`. Shouldn't the OnListEntryInit function be called at the third line of code?**

```
void ASAPI (*SetInitProc)(ADMHierarchyListRef list,
ADMListEntryInitProc initProc);
ADMHierarchyListRef listRef = sADMItem->GetHierarchyList(itemRef);
sADMHierarchyList->SetInitProc(listRef, OnListEntryInit);
ADMListEntryRef entry = sADMHierarchyList->InsertEntry(listRef, index);
```

There's a bug in ADM that prevents Init procs from ever being called for hierarchy list entries.

**Is it possible to construct dialogs without a platform resource?**

Not in the current ADM. You can define a default, empty, dialog resource and use that repeatedly.

# Glossary

## A

### Activate

When applied to edit-text items, the cursor has been set somewhere in the text, by either the plug-in/application or the user. When applied to an entry, this means the user has selected it. When applied to dialogs, this means a floating dialog has focus. The `Activate(bool)` call activates/deactivates the associated list item.

### Action

A low-level system event like a mouse-up, mouse-down, or key-down. The ADM tracker fields ADMActions. Actions can be filtered using an ADMActionMask.

### AGM

Adobe Graphics Manager. Using AGM, you can write directly to a graphics port, bypassing ADM; this is not recommended. AGM is not "exposed" to third-party developers in all Adobe applications.

### ASPoint

A point on the screen, defined by `x` and `y` coordinates. A point can appear within the coordinate system (`0,0` in the upper left, with `y` values increasing "downward" and `x` values increasing to the right) of an item, a dialog, or the screen.

### ASRect

A rectangle on the screen, defined by bottom, left, top, and right corners (each as an ASPoint). A "rect" can appear within the coordinate system (0,0 in the upper left, with `y` values increasing "downward" and `x` values increasing to the right) of an item, a dialog, or the screen.

## B

### Bounding rect

The full size of a "rect" (usually a *dialog*), including the border.

## C

### Callback

A user-supplied routine that is written and registered with ADM. A callback must follow the signature provided in the header files. Prototype signatures always are procs (end with a "proc"), but the user must define his own name.

### Clipping

An operation performed on a display element to change (decrease) its size in some way, possibly changing its shape.

## D

### Dialog

Along with *item*, the base object of ADM, usually a rectangle that appears on the screen, filled with Items. A dialog can be *modal*, *non-modal*, or *popup*.

### Dialog group

A group of *dialogs*. A *dock* is a group of panels or dialogs.

### Dividing line

The line drawn to separate list entries. This is distinguished from *separator*.

### Dock

A group of panels or *dialogs*.

### Docked panel

A panel that resides in its defined spot within a *dock* of panels.

## E

### Enable

To make an object selectable by a user. If disabled, the object is grayed out (for example, an entry may be greyed out). A disabled ADM dialog is dimmed and unusable.

### Entry

An element of a list. Not to be confused with *list entry*, which is an element in a *hierarchical list*.

## F

### Floating panel

A *dialog* that is visible by itself on the screen (i.e., not docked and part of a group of panels).

### Flyout

A type of *pop-up dialog* that appears when the user clicks on the pop-up button of another *dialog*.

### Focus

Ready to receive user input, or simply not in the background (while other items or *dialogs* are).

## H

### Hierarchy list

A list with a sublist. The sublists may in turn have sublists. An element of hierarchy lists is called a *list entry*.

## I

### Item

Along with *dialog*, the base object of ADM. An item can be a radio button, a text box with editable text, a list, etc.

### Item group

Collects several items that need to respond to calls as a group. For example, you might have five items that need to be enabled or disabled simultaneously. Once those items belong to a group, you need to enable/disable just the group.

## K

### Known

An item is in a "known" state if it has a "good" or valid value.

## L

### Leaf entry

A child in a *hierarchical list*. Regular (non-hierarchical) lists do not have leaves. An *entry* that has no *list* attached is a leaf. If the entry has a list attached it is referred to as just an *entry*.

### List

A group of *entries*. Not to be confused with a *hierarchical list*.

### List entry

An element of a *hierarchical list*. Not to be confused with an *entry*, which is an element of a regular flat list.

### Local rect

The area with the rect, usually a *dialog*, that does not include the border of the rect.

## M

### Modal dialog

A *dialog* that exists only while it is on-screen. It must be dismissed by the user before the application can resume interacting with the user. To be distinguished from a *non-modal dialog* or *modeless dialog*.

### Modeless dialog

Same as *non-modal dialog*.

## N

### Non-leaf entry

A *list entry* that is not a child.

### Non-modal dialog

A *dialog* that can exists indefinitely on-screen and must get focus before it can be used. *Docks*, *floating panels*, and *dialog groups* are examples of non-modal dialogs.

### Notification

A high-level event like an undo or redo. Not to be confused with an *action*.

### Notifier

The ADM component that fields *notifications*.

## P

### Pixel

The smallest addressable point in a display. Widths and heights are specified in pixels in ADM, as are ASPoints.

### Pop-up dialog

A special type of DIALOG that is invoked by the user, usually via a mouse click. A *flyout* is an example of a pop-up dialog. A traditional *modal dialog* can be dismissed only by the OK or Cancel button. The *pop-up modal dialog* is similar to a modal dialog in

that it stays up only while the end user is using it. The pop-up modal dialog, however, has no OK/Cancel button; it is dismissed when the end user makes a selection, presses the Esc key, etc.

**Position code**

Used with docked panels to determine which panel is first, second, third, etc., as well as where the tab is located (first, second, etc.).

**Proc**

A procedure. Often this is used synonymously with *callback*, although a proc can be invoked even though a user has not written his own callback and registered it with ADM. A common proc is the Init proc (initialization callback). Other procs include Notifier procs, Tracker procs, and Draw procs.

## S

**SuitePea interface**

Refers to the Plug-in Component Architecture (PICA), which is implemented as a series of suites. Pointers to the suites often are coded as `suiteP`; hence the term SuitePea or, sometimes, SweetPea.

**Select**

Same as *activate*, except it invalidates the entry, causing a redraw.

**Separator**

The line that goes between menu items and takes up a place in the menu list. To be distinguished from a *dividing line*.

## T

**Tool panel**

A *floating panel* with close boxes.

**Tool tip**

When the user moves his mouse over a GUI element (and does not click), a small indicator appears after some amount of (programmable) time, describing the GUI element.

**Tracker**

The ADM component that fields ADMActions.

## V

**Visible**

Appears on-screen (Mac OS) or within the application Window (Windows).

## W

**Window**

The basic element of display provided by an application.