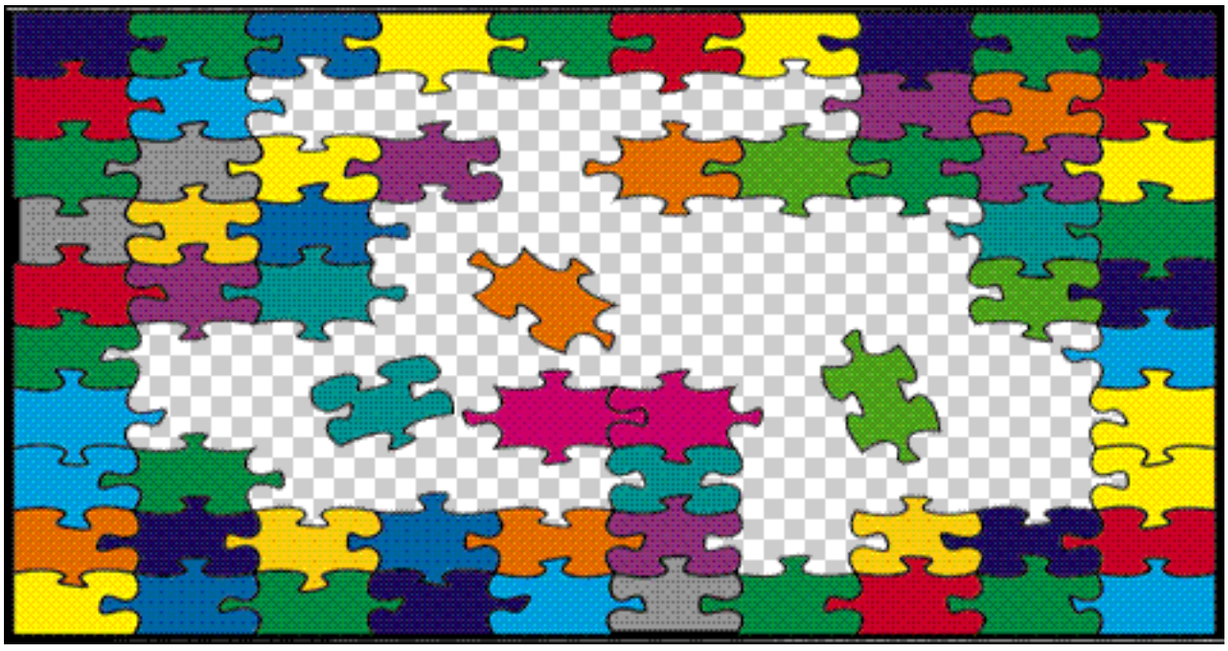
Adobe Plug-in Component Architecture
PICA



The Adobe PICA API Reference
Version 1.1 3/97

Revised: 30 March 1997

PICA API Reference Guide

Version History		
Release 1	8/1/1996	Matt Foster
Release 1.1	3/6/1997	Matt Foster and Andrew Coven

0 About This Document

This document describes the Adobe Plug-in Component Architecture, PICA. It begins with a description of the managers components and continues with chapters for each function suite. The suite chapters are in alphabetical order.

The format of the suite chapter is fairly similar with a general introduction to the suite, followed by any concepts and structures used by the suite. A description of specific functions ends the chapter.

Viewing and Printing This Document

This document was designed to be usable on the screen and on paper. When viewing on screen, a reduction of 85% will make it fit on half a 17 inch monitor. When printing, choose the Shrink to Fit option. This will give a printed page layout suitable for use in a 3-ring binder, even with double sided pages. Acrobat book marks are availabe and allow easy navigation of the file.

Conventions

Constants in this document are denoted with a preceding lowercase 'k', for instance, `kSPInterfaceCaller`. Globals variables will be of two types. General usage globals begin with a lowercase 'g', for instance, `gError`. Plug-in often keep suite references as globals. These are indicated by a lowercase 's', for instance, `sSPInterface`. The capital SP in a suite global means that the suite is provided by Suite PEA. Other hungarian notation is not used.

Accessing the Suite

Each of the suite chapters will have a section of this name with suite constants and an example of how the suites are acquired. The example will look something like this:

```
SPBlocksSuite *sSPBlocks;
error = sSPBasic->AcquireSuite( kSPBlockseSuite,
                        kSPBlocksSuiteVersion, &sSPBlocks );
if ( error ) goto error;
```

The sSPBasic variable in the code above is from the SPMessageData structure passed to the plug-in main entry point. It is assumed to be defined as:

```
SPBasicSuite *sBasic = message->d.basic;
```

Supporting Documents

There are five companion documents to this reference as follows:

Document Title	Description
AI7 Plugin Intro & Tutorial	What plug-ins are and how to write them
AI7.0 Function Reference	Reference to functions in the Adobe Illustrator API
PiPLs in Adobe Illustrator	The Plug-in Property List
The Adobe Dialog Manager API	Reference manual for the ADM API. This is the cross-platform dialog manager for Illustrator 7.0
Porting Plug-ins	Platform and Illustrator 6.0 to 7.0 porting issues

1 Intro

About the PICA Plug-in Manager

The plug-in management core of Adobe Illustrator and other applications is a technology called "PICA" (Plug-in Component Architecture). PICA provides a standard way for an application to export an application programming interface (API). In Adobe applications, code modules that use the API are called plug-ins. Host applications and plug-ins that utilize the PICA interface operate under a familiar environmnet that is flexible and extensible.

The PICA core handles the loading and calling of plug-ins, and means of exporting callback functions. This functionality is provided to both the host application and plug-ins, allowing plug-ins to provide their own APIs. In addition to a growth path for the future, PICA provides an interface for supporting legacy APIs.

There is a distinction between PICA and an application's programming interface. PICA has an API that an application uses to export its specific API. Understanding PICA plug-ins is a foundation for understanding the applications plug-in API, but little to do with the application's functionality. For instance, the Illustrator API is built on top of the application and PICA, and is used to extend the application with plug-in and art types.

This document provides a technical overview of the PICA technology. It is intended to be used in conjunction with the PICA interfaces and sample files.

(Note: The term "PICA" is used in several ways. All basically refer to the API technology or to the actual plug-in management code inside the application. They are synonymous, the architecture being interchangable with its implementation.)

The Plug-in Model

On a macro-level, the PICA plug-in model looks something like the diagram on the next page.

A host application is a normal application that incorporates the PICA plug-in architecture. Host applications implement their core functionality with an eye towards what will be necessary to expose certain functionality through the plug-in API. It uses the PICA interfaces to add it's own callback function suites to the plug-in programming interface; a group of related functions are called a "suite".

PICA plug-ins are located, loaded, and started up when the application is first launched. Plug-ins may be unloaded to free memory. Later, when the user triggers a plug-in in some fashion, will reload and call the plug-in as needed.

Plug-ins are stand alone pieces of code that interface with the host application. They will likely use some application provided functionality, but may

be completely self contained. PICA plug-ins are defined by a file containing a special resource called a Plug-in Property List, or PiPL, rather than a file type or extension. The PiPL describes to the application the PICA version for which the plug-in was written, where the code to be called is, and other information depending on the plug-in type.

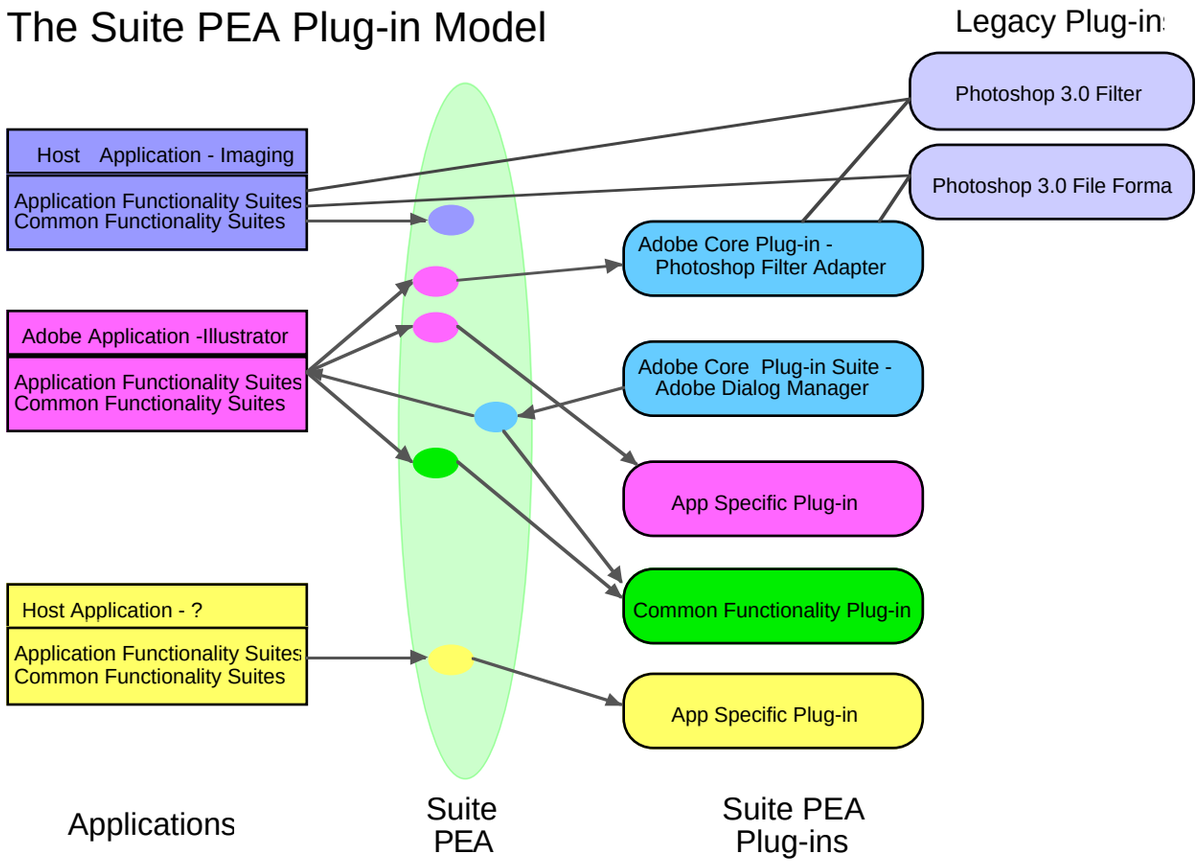


Figure 1: The Plug-in Model

All plug-ins are controlled by an *adapter*. The adapter handles all basic interfacing with plug-ins, such as the startup process and interfacing to the host application's user interface. It may also provides a suite of functions to send its plug-ins messages. PICA plug-ins are hosted by the PICA adapter, which is integral to the PICA manager.

API adapters can be added by plug-ins to implement other plug-in APIs, for instance Adobe's pre-PICA Photoshop plug-ins (dubbed "pre-suitened"). When PICA has finished loading all of the plug-ins that it knows about, it will call any adapter plug-ins to have them load their plug-ins. An adapter handles how plug-ins are added to host's user interface.

All plug-ins use the host provided API to add user interface items to the host application and access its data. For instance, they might use the tool suite to add a new tool, or the path suite to access information on bezier paths.

Plug-ins can use the PICA API to add function suites and API adapters. Plug-ins can also send messages to other plug-ins. All the PICA functionality available to the application is available to plug-ins.

In order to use a function in one of the API's suites, plug-ins first request the containing suite from PICA. If the suite is provided by another plug-in, PICA will arbitrate the situation and ensure that the plug-ins load in the correct order.

What is a Suite?

A **suite** is central to the design of PICA and a plug-in running under a host application. A suite is just a pointer to some data. PICA imposes no interpretation upon the suite's contents. It assumes the publisher of the suite and its

clients know what's inside. Typically, it is a simply a table of pointers to related functions; they might access a data type or perform some common operation. The client has the publisher's definition of the structure and makes calls to functions through the pointer. A PICA suite is analogous to a COM interface or a SOM object description.

To use a suite, it is first acquired; when it is done being used, the suite is released. To a plug-in the interface consists of just those suites that it uses and it will likely only use a subset of the many suites available. PICA will make sure that a function suite is available, reloading a providing plug-in if necessary.

Since PICA divides the whole of the interface into many small, independently versioned suites, changes to a suite become isolated and affect only those plug-ins that depend on a given suite. Likewise, new suites have no affect on existing plug-ins.

Suites are defined by a name and an api version number and are made up of a table of functions and necessary data types. The simplest of suites is the "SP Basic Suite", which looks something like this:

```
// Basic Suite defining constants
#define kSPBasicSuite     "SP Basic Suite"
#define kSPBasicSuiteVersion1

// The basic suite doesn't have any data structures

// Suite functions
typedef struct SPBasicSuite {
    SPAPI SPErr (*AcquireSuite)( char *name, long version, void
**suite );
    SPAPI SPErr (*ReleaseSuite)( char *name, long version );
    SPAPI SPBoolean (*IsEqual)( char* passedID, char* whichID );
    SPAPI SPErr (*Undefined)( void );
} SPBasicSuite;

// Errors
#define kSuiteNotFoundError"Suite not found"
```

Suites are organized around a data type or capability, such as manipulating paths or implementing a type of user interface, and ideally a suite provides all the functionality a plug-in needs to implement that capability. The PICA plug-in manager provides a handful of core suites used to implement a plug-in interface. These are used by the application to adds its interfaces and plug-ins to extend it.

Importantly, multiple versions of the same suite can exist, making it easy to grow the interface without losing compatibility with old plug-ins. A plug-in using an old version of a suite is unaware and unconcerned with the existence of the newer versions of the same suite. Because suites are small and manageable it's easier to provide old versions.

Interface Files

The API for PICA is found in a directory accompanying this document called "PICA API". The names of the files are descriptive of their contents, which in most cases is a suite to access one of PICAs data structures. The file includes the suite's identifying name and api version constants and its suite functions. If the suite has related plug-in messages they will also be defined. Descriptions of the suites are found in later chapters of this document.

Some of the API files are support files and do not have suites. These are:

```
SPConfig.h
SPHost.h
SPMData.h
SPPiPL.h
SPTypes.h
```

"SPConfig.h" contains compiler flags indicating the plug-in development platform. If part of your plug-in code is platform dependent, you may want to use these defintions:

```
#define MAC_ENV 1
#define WIN_ENV 1
```

Only one of the environment flags will be true.

"SPTypes.h" contains a few basic constants and types:

```
#define TRUE   1
#define FALSE  0
#define NULL
#define SPAPI
typedef long SPErr;
```

These constants are defined if they do not already exist on a platform. NULL is defined to be C and C++ compatible. SPAPI is used on some platforms to standardize the parameter-passing conventions. Functions on the Macintosh platform use pascal calling conventions. SPAPI on Windows is not defined as anything. All error codes in PICA are four-byte values, of type SPErr.

"SPPiPL.h" includes all the basic structures of a plug-in property list and properties. These are discussed further in the chaptert on PICA PiPLs. "SPMData.h" contains the basic message data structure passed by PICA to plug-ins. Other API files will contain message structures that build upon it. "SPHost.h" contains the interface functions used by the host to access the PICA manager. Plug-in development does not require them, but they may be of interest.

Further documentation on how PICA plug-ins are used in a host application and the inteface files for that host API may be located elsewhere in the SDK where this document was found.

Some Design Goals

The primary design goals of the PICA plug-in manager are that it is flexible enough to be implemented in multiple applications on multiple platforms; that it is capable of supporting plug-ins under those varied hosts; and that it is capable of supporting the evolving APIs of those hosts without disrupting the existing installed base of plug-ins. The PICA plug-in manager meets these goals.

To support all possible interfaces, the PICA interface is modular by design and extensible in practice. It makes its functionality available to the application and to plug-ins through “suites” of functions. Its core set of suites are used by the application to export its API and handle plug-in access. The same PICA functions allow plug-ins to add function suites, providing new functionality without affecting the core application. The ability of the host to provide adapters

A final goal of PICA was to have independence from development environments. PICA is based on the Standard C language and uses operating system services common to all platforms. This enables Adobe's commitments to its APIs on multiple platforms to be fullfilled.

Common Plug-in Interface

While Adobe's plug-in architecture design experience suggests that each host application will always provide a unique API, a common plug-in interface is still a desirable goal. Given the breath of data types and purposes of existing application APIs, such an interface is likely to evolve over time and have meaning at multiple levels. PICA serves as the starting point for the common interface.

In its first iteration, the common interface between the application and the plug-in is at the API management level and in the basic interfacing of plug-ins. The common mechanism will mitigate the learning curve for the plug-in developer, since learning how it works for one application is learning it for all. A common interface also eases the work of supporting plug-ins on multiple platforms.

Standard suites and plug-in types will extend the common interface. One category of common suites is interface and utility suites. These further ease the work of supporting plug-ins for a single application across multiple platforms.

Standard interfaces to application data and services are the next step. Plug-ins exploiting these resources will be sharable between applications. If two applications share similar data, then the suite for that data in each application could be similar or even identical. For example, bezier paths exist is several of our applications. The suite for accessing and modifying bezier paths could therefore be the same in all these applications. As a result, every bezier path plug-in can be used with all these applications.

As the number of shared interfaces expands, so will the number of common plug-ins. Issues such graceful degradation of plug-in functionality will become important, as plug-ins will try to scale their capabilities based on those of the host. Plug-ins developers will want to check for what interfaces are supported and reduce their functionality when one isn't available.

There will always be plug-ins that are meant to be used with a single application. Many exciting plug-ins will take advantage of data types unique to a host application. It follows that the suites for that data would be unique to that application and that plug-ins that require them will naturally only work with it. For example, graphs only exist in Illustrator. The suite for accessing and modifying graphs would therefore only be available in Illustrator, and as a result, every graph plug-in can only be used with Illustrator. These plug-ins would still take advantage of the common plug-in interface and likely the common utility suites.

To summarize, a PICA-based common plug-in interface is not an all-or-nothing affair, where the entire interface must be implemented or nothing will work. Nor is it an either-or affair, where it is either the Illustrator Plug-in Interface or the Photoshop Plug-in Interface. Rather, PICA is the mechanism that brings the interface (the suites) together. The interface becomes the sum of its suites, the sum (set of suites) being determined by the capabilities of the host application.

2 Plug-ins

A PICA plug-in is a piece of stand alone code that interface with a host application. The plug-in file is located in the application's plug-in folder. A plug-in has at least two properties: a resource of type 'PiPL', and a code entry point conforming to the one described below. How the plug-in interfaces to the application is defined by it being a PICA plug-in and the API exported by the host application.

Even without a host application API, a plug-in will have the behavior and functionality provided by the PICA API. This would allow it to load and unload, provide suites of its own, and manage other plug-ins, for instance, sending other plug-ins messages. The capabilities of a plug-in beyond this will primarily depend on the API extensions provided by the application. Generally there will be some interface to the application's user interface. This may be as simple as adding a menu item or something more complex such as a plug-in tool. Access to the application's data will be provided in some form.

What Defines a Plug-in?

A PICA plug-in must have two characteristics. It must have a valid Plug-in Property List ('PiPL') resource and it must have a code entry point described by the PiPL. A 'PiPL' resource contains information about the plug-in's type and how it is to be called. PICA will only consider files with PiPL resources to be potential plug-ins. The type or extension of a file is not important in this regard. Files with the correct properties as described in the next section will be added to the plug-in list.

The second characteristic of a PICA plug-in is that it must have code native to the platform on which it is to run. The entry point of the code is specified in the PiPL resource. While early plug-in APIs from Adobe would run 68k plug-ins from a PowerPC application or vice-versa, this mechanism is not provided to PICA plug-ins. The entry point of the plug-in code will be called with a variety of messages telling it actions to take.

Only plug-ins with the PICA version information in the PiPL are recognized and included in PICA's initial startup process. Non-PICA plug-ins are ignored. (See the SPAdapters chapter for information on how non-PICA plug-ins are supported.)

A plug-in will be loaded into and unloaded from memory as needed by PICA and the host application. It needs to be written assuming it will not always be kept in memory.

A PICA plug-in can expect certain services from its host. Because the plug-in may be unloaded, PICA provides it a means of storing important data when it is unloaded. Each time the plug-in is called it will be given enough information to accomplish the action to be performed. In most cases, this will include the basic suite so that it can acquire other suites.

PiPLs

The PiPL is a resource critical to PICA plug-ins as it contains the information needed to use the plug-in. As a minimum, it informs the PICA plug-in manager about the type of the plug-in, the calling mechanism for the plug-in code, and where that code exists. It may also contain other information about the plug-in. PiPL stands for Plug-in Property List and they are described fully in a later chapter.

A PICA plug-in's PiPL must contain 3 *properties*:

The '**kind**' property indicates the type of the plug-in file; it is akin to a file type. PICA will find and use plug-ins with a PiPL 'kind' of 'SPEA'.

The interface version property '**ivrs**' describes to PICA the calling conventions expected by the plug-in. It is currently 2.

A *code descriptor* property indicates to PICA where the plug-in code resides. Code descriptors are available for 68k-based and PowerPC Macintosh, and Windows based PICA plug-ins. A plug-in can have multiple code descriptor resources if is to run on several platforms. For instance a 'fat' Macintosh plug-in would have 68k and PowerPC code descriptors.

If a plug-in exports one or mores suites of functions it must have a fourth property to indicate this. The export property, 'expt', lists the names and api version numbers of the suites that a plug-in provides. When a request for an external suite is made, PICA will use the export PiPL to find the providing plug-in. When this has been done, the now available suite reference will be returned to the requesting code.

The kind, version, and code descriptor properties described above should be in a standard platform resource. Examples are provided with the source code accompanying this document. PICA also provides a means of creating resources at runtime.

Plug-in Loading Order

The loading order of plug-ins becomes important when one plug-in depends on a resource provided by another, as the resource providing plug-in must be loaded first. To ensure that the interdepencies of plug-ins are handled correctly, PICA plug-ins that provide a resource such as a suite declare in advance what they export. PICA will use this information when loading and executing plug-ins, ensuring that suites and other resources are available. The export information is declared in the PiPL resource.

The Plug-in Entry Point

The compiled plug-in code referenced by the code descriptor is written in C. PICA communicates with your plug-in by calling its entry point (e.g. main( )), which is defined by the platform code descriptor in the PiPL. Here's what and entry point looks like:

```
#if Macintosh
SPAPI SPErr main( char *caller, char *selector, void *data ) {
#else
SPAPI SPErr PluginMain( char *caller, char *selector, void *data ) {
#endif

    SPErr error = kSPNoError;
...
    return error;
```

```
}
```

Three arguments are passed to the main( ) routine regardless of the reason it is called; together they make up a *message*. The first two are C style strings indentifying the message action, or what the plug-in is supposed to do. The third is a pointer to a data structure. It is undefined at main( ) because it's content depends on the message action received. When you determine the message action, you will type cast the data as needed. The result of main( ) is an 4-byte error code.

Message Actions: Callers and Selectors

The message action passed to plug-in consists of two identifiers, a caller and a selector . The caller indicates the sender of the message (PICA, the host application, or a plug-in) and a general category of action. The selector defines the action to take within that category. For instance, an application might send a message action based on these two indentifier strings:

```
// These are the hypothetical callers and selector
#define kMyAppMenuCaller      “My App Menu”
#define kMyAppGoMenuSelector  "Go Menu"
```

The caller and selector identifiers are C style strings. This is so that new message actions can be easily defined by other applications and plug-ins with little chance of conflict. The convention used is for the caller to have a name or abbreviation of the caller's application or company at the start of the string. For instance, all PICA action identifiers begin with 'SP'.

PICA message actions use four callers with many associated selectors. The standard PICA callers and selectors are described further below. For the message caller, selector, and data associated with a host application API, see that application's API documentation.

Whenever a plug-in is loaded into memory or unloaded from memory, PICA will send it an access message. The action will be the access caller and a reload or unload selector. This is the plug-ins opportunity to setup, restore, or save state information. The access caller/selectors bracket all other callers and selectors. The access/reload will be the first message received. The access/unload will be the last message received. A plug-in should not acquire or release suites other than those built into PICA at this time.

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

PICA has three interface actions, where the plug-in can interact with its host. The host application will supplement these with its own interface-like actions. When the application is first launched PICA will startup all the plug-ins it finds. This is an opportunity for the plug-in to allocate global memory, add user interface items to the host application, and do other initialization. A plug-in adding components to PICA, such as suites and adapters, would do it at this time. The startup message action is received at this time: the interface caller and the startup selector.

When the user quits the host application, PICA tells all of its plug-ins that it is shutting down. It will send each a shutdown message action, consisting of an interface caller and shutdown selector. Shutdown is intended for flushing files and preserving preferences, not for destruction. A plug-in that exports a suite should not dispose its PICA globals or suite information, since it may be called after its own shutdown by another plug-in's shutdown. For example, if your plug-in implements a preferences suite that other plug-ins

use, they may call you in their shutdown handlers after you've already shut down.

A third message action received by a PICA plug-in is an opportunity to display information about a plug-in on the screen. The action is indicated by the interface caller and the about selector.

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

The remaining PICA message actions are used by plug-ins to specify properties at runtime and by plug-ins that add API adapters. More information on using these selectors is in the related chapters.

Handling Callers and Selectors

A PICA plugin's organization is largely based on the messages received by its main( ). The main routine of a plug-in basically becomes a switch implemented as a series of string compares that call functions appropriate for the message action caller and selector. For instance:

```
SPAPI SPErr main( char *caller, char *selector, void *data ) {

    SPErr error = kSPNoError;

    if ( strcmp( caller, kSPAccessCaller ) == 0 ) {
        if ( strcmp( selector, kSPAccessReloadSelector ) == 0 )
            error = MySetupGlobals( data );
        else if ( strcmp( selector, kSPAccessReloadSelector ) == 0 )
            error = MySetupGlobals( data );

    } else if ( strcmp( caller, kSPInterfaceCaller ) == 0 ) {
        if ( strcmp( selector, kSPInterfaceStartupSelector ) == 0 )
            error = MyStartupPlugin( data );
        else if ( strcmp( selector, kSPInterfaceAboutSelector ) == 0
)
            error = MyAboutPlugin( data );

// check for each of the other callers and selectors...
    else if ( strcmp( caller, kMyAppMenuCaller ) == 0 &&
         strcmp( selector, kMyAppGoMenuSelector ) == 0 )
        error = MyHandleMenu( data );
//  else if ...
    return error;
}
```

Message Data

The other argument passed to the plug-in entry point is a pointer to a message data structure. This structure contains enough information for the plug-in to handle the message. For instance, if the message action was that a mouse was clicked, the message data structure would contain the mouse position.

The contents of the message data structure depend on the message action and are not completely known until the plug-in has identified this. Most PICA messages will have a minimum set of information embedded in the structure, the SPMessageData structure at the beginning of the message data structure. This is actually dependent on the type of plug-in as implemented by the host application. For instance, Adobe Illustrator plug-in types

expect that the message data is embedded. Other hosts may define messages where it is not guaranteed.

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

typedef struct {
     SPMessageData d;
} SP<i>caller</i>Message;
```

A host application message will likely have this core message data embedded, but it is possible that it is not provided. Check the host application's API documentation for more information.

The core of any plug-in message data is set by PICA and includes the basic information for the plug-in to operated. The *self* member is a reference to the plug-in being called. The *globals* member is a 4-byte value specified by and preserved for the plug-in. The *basic* member is a reference to the PICA Basic Suite.

The reference to the running plug-in's *self* is used to add plug-in suites and adapters to PICA and other plug-in data to host application. PICA and the host application will store this value with the added data. It will then be used to recall the plug-in as needed.

The plug-in can set the *globals* member to any four byte value; most likely this will be a pointer to a block of memory allocated by the plug-in. This value will be preserved by PICA when the plug-in is unloaded and passed back to the plug-in each time it is called. Plug-ins use this block to store any state information they need to maintain.

The SPBasic suite reference *basic* contains functions with which the plug-in can acquire and release any other suites it needs to run.

The message data structure received by a plug-in when the message action's caller is `kSPInterfaceCaller` is an SPIntefaceMessage structure. This is simply a container for the SPMessageData:

```
typedef struct SPInterfaceMessage {
    SPMessageData d;
} SPInterfaceMessage;
```

The message data structure received by a plug-in when the message action's caller is `kSPAccessCaller` is an SPAccessMessage structure. This is simply a container for the SPMessageData:

```
typedef struct SPAccessMessage {
    SPMessageData d;
} SPAccessMessage;
```

When the host application or a plug-in wishes to send a message to a plug-in, it will pass in a relavant message data structure. The menu action above might have a message structure such as this:

```
typedef struct {
     SPPluginData  d;
     long          myMenuCommandID;
} MyAppMenuMessage;
```

It includes the SPPluginData at the beginning of the structure followed by the information menu command to be handled. The host application API documentation describes its message actions and data.

Using Suites and Callback Functions

An application's API is composed of callback functions organized into suites. Before a plug-in can use a function that is part of a suite, the suite containing it must first be *acquired*. A function suite is an structure filled with function pointers and when a plug-in acquires a suite, a pointer to this structure is returned.

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

Acquiring and Releasing Suites

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

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

A plug-in uses the first function, `AcquireSuite( )`, to gain access to a suite of functions. All acquired suites must be released with the `ReleaseSuite( )` function when the suite is unneeded.

To acquire a suite, you first need to declare a suite pointer. For instance, suppose a host application provides the following suite with name and version constants:

```
#define kMyAppMenuSuite           "My App Menu Suite"
#define kMyAppMenuSuiteVersion    1

typedef struct MyAppMenuSuite {
    AddMenu( SPPluginRef plugin, MyMenuRef *menu );
}
```

The plug-in's reference to the above suite would be:

```
MyAppMenuSuite *sMyAppMenu;
```

A suite is acquired and released using its name and version number, found in its published header file. So to acquire the above menu, you would do something like this:

```
SPErr error;
error = (*message)->d.basic->AcquireSuite( kMyAppMenuSuite,
                                    kMyAppMenuSuiteVersion, &sMenu );
```

A pointer to the acquired suite's functions is returned in sMenu.

Calling a Suite Function

The example above calls a function in the basic suite. Functions in other suites are called using a pointer to an acquired suite. For instance, the above

variable *sMenu* points to the structure with the hypothetical menu suite function pointers.

The suite function pointer is dereferenced and used as a function pointer using the calling form:

```
sMenu->function();
```

Since they are used throughout the plug-in code, it is convenient to make suite variables global. The convention used for these global variables is a small ‘s’ followed by the suite name, e.g. sBasic as shown above, sMenu for the menu suite, etc.

A Complete Example

Suppose a plug-in is going to add a menu item to the host application. To do this it must use the AddMenu( ) function, which is a part of the “My App Menu Suite”, version 1. The suite name and version number are passed to the AcquireSuite( ) function. After the function is used, the suite is release. This whole process is demonstrated below.

```
SPBasicSuite *sBasic = message->d.basic;
MyAppMenuSuite *sMyAppMenu;
SPErr error;

// request the needed suite
error = sBasic->AcquireSuite( kMyAppMenuSuite,
                              kMyAppMenuVersion, &sMyAppMenu );
if ( error ) goto error;

// use the suite
error = sMyAppMenu->AddMenu ( message->plugin, nil );

// release the suite
sBasic->ReleaseSuite( kMyAppMenuSuite, kMyAppMenuVersion );

error:
    handleError( error );
```

Suite .h Files

Every suite will have the suite name and version in the suite header file as well as other definitions, such as error strings, particular to their function. If the suite defines plug-in messages they will also be found in the header file. At the end are the suite functions. The function pointers should be fully prototyped.

Platform Considerations

While Adobe PICA plug-ins are highly portable across platforms, there are a number of platform considerations you will need to keep in mind when writing them. Differences are related to the architectures of the hardware or operating system on which host is running, and these have been abstracted so that the API call works on both environments with a minimum of platform support code. General issues are discussed below.

The main cross platform differences in plug-ins will often be in presenting the user interface and resource data; depending on user interface complexity, this can be a significant undertaking. Other differences involve how the host allows access to its data types and how supported plug-in types are added to the application. The Adobe PICA manager does not provide

resources to handle this situation, though the host may. This host API should also be cross platform, though there may be exceptions. Some Adobe Systems applications support the Adobe Dialog Manager API (ADM), which is itself an Adobe PICA plug-in. For more information on how the plug-in host handles cross platform issues or on ADM, check the documentation and sample code accompanying this SDK.

Plug-in Property Lists

All Adobe PICA plug-ins must have a plug-in property list (PiPL), which is a resource used to identify compatible plug-ins and provides minimal information about them. Complete information on the structure of a PiPL resource is found in the "Adobe PICA PiPLs" chapter. Resource files with minimal PiPL data are included with the various sample projects, and can be used with your plug-in project. The file called "BasePiPL.x", where x is a platform resource extension. The basic PiPL has three properties: a type, a version number, and a code descriptor.

Table 1: Adobe Illustrator 7.0 Plug-in Properties by Platform

Property/Platform	Macintosh 68k	Macintosh PowerPC	Windows 32
Type	SPEA	SPEA	SPEA
Version	2	2	2
Code Descriptor	'm68k'	'pwpc'	'wx86'

Additional property messages, notably "export" properties, are described in elsewhere in the plug-in SDK.

The Entry Point

All plug-ins have a single entry point to the plug-in code. On the Macintosh 68k platform, this is of type pascal and must be the C code main( ) function. On the PowerPC, it can be any function name, though to minimize code differences, main( ) is also used.

```
main( char *caller, char *selecter, void *message );
```

On Windows, the entry function is by convention PluginMain( ).

```
pluginMain( char *caller, char *selecter, void *message );
```

For PowerPC and Windows, the actual entry point is specified with the PiPL resource code property for the running platform.

On Windows platforms, this entry point is also declared as a .dll function in the .def file for the plug-in. This is the only function that must be exported. In addition to the plug-in entry point, a standard main routine for handling the .dll needs to be created. You could use this, for instance, to store a copy of the .dll instance in a global. DLL support files are included in the sample code and can simply be copied and reused.

Memory

Adobe PICA provides the Block suite for pointer based memory allocation. This is linked to the host application's memory manager. The host application may provide other memory management function suites, for instance to provide a single entry to machine specific memory calls (Handles on the Macintosh and HANDLEs on Windows).

Resources

When a Adobe PICA plug-in is running on the Macintosh, its file's resource fork is the current one. Adobe PICA plug-ins for Windows keep standard resources in the plug-in file. To access these, you will need to have the

HInstance of the plug-in file. This can be obtained by using the provided SPAccessSuite function,

```
SPAPI SPErr GetAccessInfo(SPAccessRef access, SPPlatformAccessInfo
*info );
```

The host application may provide some cross-platform means of accessing resources, for instance, the Adobe Dialog Manager allows access to string list and picture resources, APIs.

Byte Information and Structures

There are actually very few Adobe PICA data types. Specific data structures for the Macintosh and Windows implementations are usually the same with the exception of platform dependencies such as byte order and alignment. On Windows byte alignment is to 4 byte boundaries. On Macintosh, structures are 68k 4-byte aligned. This is handled in the header files, so the plug-in project can have different internal alignment.

System Requirements

For Macintosh, both 68k and PowerPC platforms are supported. The plug-in must have a native version for the platform on which it is to run (they must be fat; 68k plug-ins do not run on the PowerPC). For Windows, plug-ins should be created for a Win32 platform such as Window NT or Windows95.

Project Considerations

For Macintosh plug-ins, use a 68k and PowerPC project. Build the 68k project into an RSRC file and then include this in the PowerPC project. The final output can be of any type and creator.

For Windows plug-ins, use a make file or project to create a .dll file. The extension of the file is unimportant as a plug-in's PiPL resource identifies it. The host may have a standard extension for it's plug-ins, e.g. Adobe Illustrator's default extension is ".aip".

Table 2: Project Informatin by Platform

Project/Platform	Macintosh 68k	Macintosh PowerPC	Windows 32
Project Type	Code Resource defined by PiPL	Shared libary	.DLL
Default File Type/ Extension (Example is Adobe Illustrator)	Creator - 'ART5' Type - 'ARPI'	Creator - 'ART5' Type - 'ARPI'	'.AIP'

3 PiPLs

The Plug-in Propery List Resource

A Plug-in Property List, commonly refered to a ’PiPL’ (pronounced pipple), resources provide a host application information about a plug-in. This information includes indicators about the types and locations of available code, versions, and other dependencies of the plug-in.

PiPLs were first used in Adobe Photoshop 3.0 plug-ins. They have been adapted for use with the PICA by ignoring certain Photoshop specific properties and defining others. The general PiPL definition is the same as that for Photoshop. More information on PiPL resources can be found in that SDK’s documentation. While the complete definition of of the PiPL structure is useful to have, the last sections of this chapter describe what you need to get quickly started with the PiPLs.

The PiPL Structure

Any plug-in property list has a version number and a count followed by a sequence of arbitrary length byte containers called properties. A ‘C ‘ struct definition for the plug-in property list is:

```
typedef struct PIPropertyList
    {
        int32 version;
        int32 count;
        SPProperty properties[1];
    } PIPropertyList;
```

Table 1: PiPL Fields and Purposes

Field	Purpose
version	Denotes the version of this specification to which the 'PiPL' is formatted. The current version is 0.
count	Holds the number of properties contained in the 'PiPL'. 0 is a valid value denoting a 'PiPL' with no properties.
properties	A variable length array of variable length property data structures. Holds the actual contents of the 'PiPL'.

Properties

Each property has a vendor code, a key, an ID, a length, and property data the size indicated by the length.

The ‘C’ struct definition for the plug-in properties is:

```
typedef struct PIProperty
    {
        OSType vendorID;
        OSType propertyKey;
        int32  propertyID;
        int32  propertyLength;
        char   propertyData [1];
    /* Implicitly aligned to multiple of 4 bytes. */
    } PIProperty;
```

The fields are defined as:

Table 2: Property Fields and Purposes

Property Field	Purpose
vendorID	Identifies the vendor defining this property type. This allows other vendors to define their own properties in a way that does not conflict with either Adobe or other vendors. It is recommended that a registered application creator code be used for the vendorID to ensure uniqueness. For instance, all Photoshop properties described in the Photoshop Plug-in SDK document use the vendorID '8BIM'. All PICA properties use the vendorID 'ADBE'.
propertyKey	Specifies the type of this property. Property types used by Photoshop are documented below. (Think of a property type as similar to a resource type.)
propertyID	In theory this can be used to store more than one property of a given type (rather like a resource ID). In practice, this field is always zero. It should be thought of as reserved for future use.
propertyLength	Contains the length of the propertyData field. It does not include any padding bytes after propertyData to achieve four byte alignment. This field may be zero.
propertyData	A variable length field that contains the bytes which are the contents of this property. Any values may be contained.

All list and property fields are four byte aligned.

Platform Dependencies

The 'PiPL' format is fairly portable in that everything is four byte aligned. All OSType and int32 fields are represented in native byte order for a given platform so the bytes of "the same" 'PiPL' will differ between a big-endian machine (e.g. the Macintosh) and a little-endian machine (e.g. an Intel x86 based Windows machine). If one examines the bytes of the PiPL section of an x86 resource binary, it will be backward compared to the Mac. If you use the pre-defined PI-types, they will be interpreted and stored correctly as in the following example (see PIKindProperty). If, however, an OSType has not been defined and you wish to enter it as a 4-char series, then (since it is not interpreted as a long) you would have to supply the chars in reverse order.

'PiPL's are intended to collect all plug-in metadata in a single place. For cross platform development, this will be quite useful since MS-DOS and its Windows derivatives lack a resource management mechanism. Plug-in developers other than Adobe are encouraged to define new properties for extensions to plug-in metadata rather than introducing new resource types.

Types

The following types will be used in defining properties:

Table 3: Types Used in Properties

Type	Definition
int16, int32	These are 16 and 32 bit integers respectively. They are stored within the 'PiPL' in native byte order.
OSType	Same representation as an int32 but tyPICAlly denotes a Macintosh style 4 character code like 'PiPL'.
PString	A Pascal style string where the first byte gives the length of the string and the content bytes follow.
CString	A C language style string where the content bytes are terminated by a null character.
Structures	Structures are tyPICAlly represented the same way they would be in memory on the target platform. Native padding and alignment constraints are observed.
Arrays	Arrays are represented as a contiguous set of entries in the 'PiPL' tyPICAlly with native padding and alignment constraints observed. The length of the array is usually determined by the property length for arrays of fixed length structures or types.

General properties

The following are property keys recognized by PICA.

Plug-in Kind **OSType**

```
#define    PIKindProperty    'kind'
```

This property encodes the type or kind of a plug-in. The expected value for a PICA plug-in is 'SPEA'. Illustrator 6.0 used a 'kind' property 'ARPI'. Photoshop defines other valid values such as a filter being '8BFM'. These values, while valid, are not recognized by the PICA loader; they may be recognized by a PICA adapter.

Version of kind specific API **int32**

```
#define    PIVersionProperty  'ivrs'
```

This property indicates which revision of the plug-in interface expected by the plug-in. The number indicates mechanisms such as the form of messages to be passed, and a version change should be assumed to be 100% incompatible with other versions.

Code Descriptor Properties

Code descriptors tell the application running a plug-in the location of a plug-in's code. More than one code descriptor may be included to build a "fat" plug-in which will run on different types of machines. PICA does not support emulated plug-ins, so if a code descriptor for the running platform does not exist, the plug-in will not be loaded. Function pointers in PICA suites are to code for the platform on which it is running. For PowerPC code this means native function pointers will be provided and that routine

descriptor operations are not required either in calling the plug-in or for the plug-in to invoke PICA or host application callback functions.

68k code descriptor — PI68KCodeDesc

```
#define PI68KCodeProperty     'm68k'
```

This descriptor indicates a 68K code resource. The type for this property is:

```
typedef struct PI68KCodeDesc {
        OSType  resourceType;
        int16   resourceID;
    } PI68KCodeDesc;
```

Any resource type may be used. For instance, the convention for PICA plug-ins running under Adobe Illustrator 7.0 is 'ARPI', number 16000

PowerPC code fragment descriptor — PICFMCodeDesc

```
#define PIPowerPCCodeProperty 'pwpc'
```

This descriptor indicates a PowerPC code fragment in the data fork of the plug-in file. The type for this property is:

```
typedef struct PICFMCodeDesc {
        long    fContainerOffset;
        long    fContainerLength;
        char    fEntryName[1];
    } PICFMCodeDesc;
```

The fields are defined as follows:

Table 4: PICFMCodeDesc Fields and Descriptions

Field	Description
fContainerOffset	Contains the offset within the data fork for the start of this plug-in's code fragment. This allows more than one code fragment based plug-in per file.
fContainerLength	Holds the length of this plug-ins code fragment. If the fragment extends to the end of the file (e.g. it is the only fragment in the file), the container length may be 0.
fEntryName	The entrypoint name is represented as a Pascal string and is used to lookup the address of the function to call within the fragment. If the entrypoint name is a zero length string, the default entry-point for the code fragment will be used. The entrypoint name allows a single code fragment to contain more than one plug-in. (Note: in order for the Code Fragment Manager to find an entry-point by name, that name must be an exported symbol of the code fragment.) Entry point names allow more than one plug-in to be exported from a single code fragment.

Windows 32-bit DLL code descriptor — PIWin32x86CodeDesc

```
#define PIWin32X86CodeProperty  'wx86'
```

This descriptor is used for 32 bit Windows DLLs. The type for this property is:

```
typedef struct PIWin32X86CodeDesc {
    char fEntryName[1];
} PIWin32X86CodeDesc;
```

The fields definitions are:

Table 5: PIWin32x86CodeDesc Fields and Descriptions

Field	Description
fEntryName	The entry point name is used to lookup the function which is called to invoke the plug-in. The name is represented as a NULL terminated string. The string may need to be padded with additional NULL charcters to satisfy the 4 byte alignment requirement..

Export Properties

Export properties are used by plug-ins to inform PICA of the suites and possibly other resources which they provide. Exports allow interdependencies between plug-ins to be resolved by PICA in a logical manner. A plug-in should have at least one export property, even it is simply the name of the plug-in or a dummy string.

For instance, if one plug-in needs a suite provided by a second plug-in, the second plug-in's export property insures that PICA will be able to find and load the second plug-in before the first one. The plug-in exporting the suite is able to initialize its function lists and other values. The correct loading order allows the first plug-in's suite request to succeed.

The simplest case of plug-in dependencies is with function suites as described above. Other dependency information can be indicated by an export property. On a broader scope, plug-ins may depend on the existence of another plug-in even if it doesn't explicitly export callback functions. A possible example of such an exported property would be a menu item provided by another plug-in.

Export properties are defined below.

Exports List Property Descriptor **`PIExportsList`**

```
#define PIExportsProperty     'expt'
```

This descriptor contains the list of dependencies that a plug-in exports:

```
typedef struct PIExportDesc
    {
        long                fCount;
        PIIEListsDesc       fExports[1];
    } PIExportDesc
```

The fields definitions are:

Table 6: The PIExportDesc Fields and Descriptions

Field	Description
fCount	The number of suites (resources) exported by the plug-in.
fExports	A variable length list describing the suites provided by the plug-in.

PIIEListsDesc

The *fExports* field of the PIExportDesc structure contains a list of exported resource descriptions. These definition of the structure is:

```
typedef struct PIIEListDesc
{
    long    fLength
    CString fName; // padded to four bytes
    long    fVersion;
} PIIEListDesc;
```

The field definitions are:

Table 7: The PIIEListDesc Fields and Descriptions

Field	Description
fLength	The total length (including 4 bytes for this field) of the AIIEListDesc record.
fName	A C style string with the name of the suite to be exported. The usable names of suites are found in the API documentation and header files.
fVersion	The version of a suite provided. Supported versions should also be listed in the API documentation.

Dynamically Declared Properties

PICA supports resource PiPLs and certain properties *must* be specified staticly in a platform resource. PICA also provides a mechanism for declaring properties in at runtime.

PICA will find resource based properties first and this can be used to optimize processes such as startup. Providing a property in a resource allows PICA to obtain it without loading and calling the plug-in. If a plug-in does not export any suites or resources, it should include a single resource-based export. This can be the name of the plug-in or a dummy string.

If the ‘expt’ or other property is not found in a plug-in’s PiPL resource, the plug-in will be sent two messages requesting the property. The plug-in can build the appropriate property information and return a pointer to it. or it may ignore the request. In either case, PICA adds the new (possibly NULL) property to the plug-in’s property list. Subsequent searches for the property will find the stored version (that is, it only asks the plug-in once for a particular property).

The message action is composed of a properties caller and two selectors:

```
#define kSPPropertiesCaller           "SP Properties"
#define kSPPropertiesAcquireSelector  "Acquire"
#define kSPPropertiesReleaseSelector  "Release"
```

and they are received at the plug-in’s entry point just as any other message. Within the main( ) function, you would determine the message type:

```
FXErr main( char *caller, char *selector, void *data ) {
    if ( strcmp( caller, kSPPropertiesCaller ) == 0 ) {
        if ( strcmp( selector, kSPPropertiesAcquireSelector ) == 0 )
{
```

```
            error = AcquireProperty( data );

        else if (strcmp(selector, kSPPropertiesReleaseSelector) == 0
)
            error = ReleaseProperty( data );

    else
        // process any other messages
}
```

and then call a routine to create or release the property structure. When creating the PiPL in memory, your should use the PICA Blocks Suite to allocated the block of memory required.

The data passed with these message actions is:

```
typedef struct SPPropertiesMessage {
    SPMessageData d;

    PIType vendorID;   // same as PiPL definition
    PIType propertyKey;// same as PiPL definition
    long propertyID;   // as always, 0

    void *property;    // return the property here
    long refCon;       // for plug-in's use. Set on acquire,
                       //  given back on release
    long cacheable;    // most likely true

} SPPropertiesMessage;
```

When the `kAISelectorAcquireProperty` message is received the **`vendorID`** and **`propertyKey`** fields define the requested property. PICA only requests 'ADBE'/ 'expt' properties at runtime; the host app may request others. The **`propertyID`** field is 0, as defined in the preceding PiPL description.

Based on the request, the plug-in must create the property in memory exactly as defined in the PiPL description and return a pointer to this memory block in the **`property`** field.

set the **`cacheable`** field to true if the information in the property data will not change. Cacheable properties may be stored by the host in a startup preferences file. If a property can change between sessions of the application, the field would be set to false.

When the `kAISelectorReleaseProperty` message is received, the plug-in should free the memory allocated to create the property.

Working with PiPLs

Basic PiPL resources usable by most plug-ins are found in the SDK in a folder call "Base PiPLs". "Most plug-ins" are those that do not export a suite. These PiPLs cans simply be included in your plug-in project file.

The Macintosh resource will support fat plug-ins and is a file called "BasePiPL.rsrc". It describes a plug-in with 68k code in the resource ARPI, 16000; with PowerPC code in the data fork; and with no exports.

The Windows resource will support a plug-in with no exports written for a Win 32 Intel-basd platform. It is found in a file called "BasePiPL.RC". The plug-in entry point specified by it is `PluginMain( )`.

On Macintosh the easiest way to edit a PiPL is using the program Resorcerer. A Resorcerer template in the file "BasePiPL.rsrc" makes editing property lists straightforward. (ResEdit resource templates cannot handle a resource as complex as a PiPL.) Windows PiPLs are currently edited in a text file.

Examples of plug-in allocating PiPLs at runtime are included in the host SDK. PiPL resource files equivalent to those above, but without an 'expt' property are found in the "Base PiPLs" directory as"NoExptPiPL.RC" and NoExpt-PiPL.rsrc".

4 The Core

Based on the preceding chapters, there are several elements that make up the PICA architecture that need to be described further: three components and two interfaces. The host application interfaces to the PICA library and uses it to implement and export its API. PICA then handles interaction with the plug-ins, loading them and allocating function suites as needed. Other interactions, for instance a plug-in making calls directly to the application using an application defined interface, are handled by the standard suite mechanism and are not described here.

Internal Data Structures

The PICA manager is added to an existing application to handle the plug-in interface. PICA is largely self contained, relying on only a small number of functions provided by the host. It contains all the data structures and code to handle scanning for plug-ins and loading and calling plug-ins designed for it. It provides a core set of suites needed to use and extend the PICA plug-in model, and arbitrates access to its suites and other function suites added by multiple sources.

PICA is built upon four internal lists:

The first list kept by PICA is the *adapter list*. An adapter handles the interfacing of a plug-in to the host application. PICA plug-ins are hosted by an internal PICA adapter. The host application and plug-ins can add other adapters to the list, allowing non-PICA plug-in to run under the PICA API. The adapter will search the file list for plug-ins types that it supports and add them to the plug-in list. When notified by PICA to do so, the adapter is reponsible for loading and calling the plug-ins it adds to the host, and must do any conversion of messages, data structures or other API elements. There will always be at least one adapter in the adapter list, PICA's internal adapter.

The *file list* contains every file that is a potential plug-in. The PICA plug-in manager searches some folder or directory for plug-ins; the file list is basically a flattened list of all files in that directory and any subdirectories, including folder aliases on the Macintosh. Using it avoids walking the folder hierarchy repeatedly and ensures consistent behavior. Each file entry in the list has platform specific information about it's location and attributes.

PICA's internal plug-in adapter use the file list to scan for plug-ins. It makes the list available through a standard suite so that the application or adapter plug-ins can also use it. Since it indescriminately adds all the files in the indicated directory, it is likely that some entries will not be plug-ins. This is intended so that the host application and plug-ins can also use it to check for the existence of support files, such as dictionaries or a start up files. Plug-in API adapters should use it search for other plug-ins. In the diagram below, the plug-in list entries marked PS were added from the file list by the PS Adapter, a plug-in.

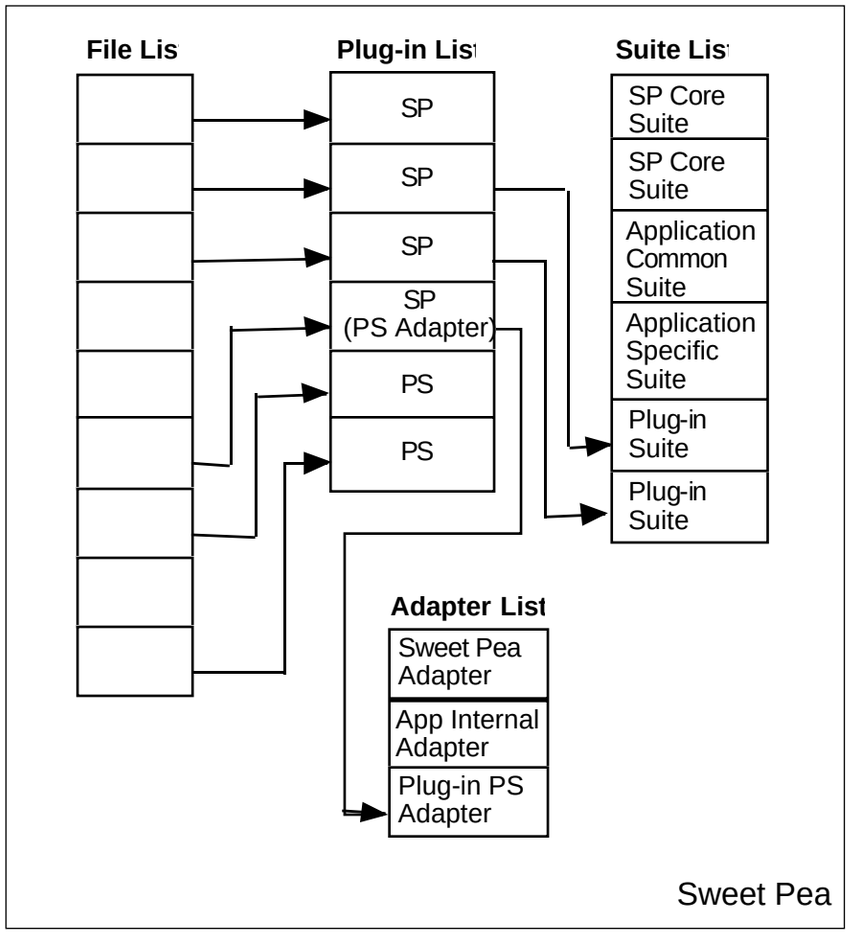


Figure 1: Internal Data Structures

The *plug-in list* is a subset of the file list. It contains references to files that are known to be plug-ins. Entries in the plug-in list have a reference to the plug-in's file, attributes, and a reference to the adapter supporting the plug-in. After PICA has completed building the files list, its internal adapter iterates through the list and checks whether or not each file is valid PICA plug-in. It adds it plug-ins to the plug-in list with their host adapter set to the PICA adapter identifier.

As with the file list, PICA provides a suite of functions to access the plug-in list. If a plug-in implements an adapter for a non-PICA plug-in, it should scan the files in the file list and add its plug-ins to the plug-in list with an appropriate adapter identifier. PICA keeps the master list of all plug-ins. When it is time to call a non-PICA plug-in in this list, the caller can determine the host adapter to call it appropriately.

The final list internal to PICA is the *suite list*, where it keeps track of all available function suites and such information as who provides a suite's functionality and the access count. When PICA is first initialized, it places its core suites in this list, including the "Suites Suite". The "Suites Suite" is used by the application to add its Adobe common and application specific suites to the suite list. A plug-in uses the same mechanism to extend the API. When a plug-in requests some API functionality by acquiring a suite, PICA scans this list to determine if it is available and who provides it. If the requested suite is from a plug-in, PICA loads the providing plug-in into memory before making the suite available.

One additional data structure used and made available by PICA is a string pool. The string pool is an efficient storage space for C style strings. When a string is placed in the pool, PICA first checks to see if it already exists. If not, the string pool manager will copy the string into the string pool and return a pointer to it. If the string already exists, a pointer to the existing string will be returned.

Suites and Data Structure Interfaces

Each of the PICA data structures is accessible to the host application and its plug-in through a standard suite. A small number of other suites is made available to provide other functions useful for managing plug-ins and by plug-ins.

The *files*, *adapters*, and *plug-ins* suites are fairly similar. Each has functions for creating and disposing their list type. New elements can be added to the list; the list can be traversed; and information on the entries such as file information can be retrieved. The plug-ins suite has a number of associated suites: *access*, *interface*, and *properties*. The access suite handles the loading and unloading of a plug-in's code. The interface suite is used to send a plug-in a message. When PICA wants to load and run a plug-in's code it uses these suites similar to this pseudo-code example:

```
access->load plug-in
interface->set up plug-in message
access->call plug-in
interface->save plug-in data
access->unload plug-in
```

The properties suite is used by PICA to keep track of the properties of a plug-in. Properties are kept in lists similar to the main PICA data lists, but each plug-in has a seperate list of its properties. If PICA, the host app, or a plug-in needs to examine the the elements of a plug-in's PiPL for some reason or find a particular property, it would use the functions in this suite.

The last suite used to manage plug-ins is the *caches* suite. If a low memory situation is encountered and plug-ins currently cached in memory need to be unloaded, the function in this suite can be used. The suite's single function will unload an unused plug-in from memory. It can be called repeatedly until sufficient memory is available or all possible plug-ins have been unloaded. It is not likely that plug-ins will use this suite.

The *string pool* suite is similar to the list management suites in that it can accesses an internal data structure. It does not need the iterator functions or information functions of these suites, so it is much simpler.

There are two suites for accessing PICA's suite list. The *basic* suite is the simpler of the two, providing two suite access functions. It is primarily how plug-ins acquire and release the interfaces made available by PICA. The *suites* suite gives complete access to the suite list in the same manner as file and adapter suites give access to their lists. New suites can be added; the list can be traversed; and information such as the provider of a function suite can be obtained. Plug-ins exporting suites will use the suites suite to make their APIs available.

While the data structure suites provide functions to create new lists or string pools, PICA keeps its central lists, and these are the main structures that will be used. To obtain a reference to any of PICA data structures the *runtime* suite is used. The suite provides functions to access PICA's string pool, suite list, file list, plug-in list and plug-in adapter list references.

The last of the standard suites provided by PICA are the block suites. The *blocks* suite is a simple pointer based memory manager, similar to the standard C libary malloc( ) and free( ) routines. Plug-ins must use this in some cases when allocating blocks of memory. During the development period of a host application, the host can have PICA track debugging information on blocks allocated with the blocks suite. When a request is made for memory through the blocks suite, PICA tags the blocks with a string passed to the

block allocate function. If available, this information is accessed using the *block debug* suite functions.

List Management

PICA data structures are kept in lists. All PICA lists are used in much the same manner. A general discussion is given here in lew of a slightly varying description in each suite chapter; exceptions to the process outlined here will be given as needed.

PICA data structures and the lists in which they are accessed by an opaque reference. The list is made up of references to a data type. To traverse the list, a list iterator is used.

```
typedef struct SPDataType *SPDataTypeRef;
typedef struct SPDataTypeList *SPDataTypeListRef;
typedef struct SPDataTypeListIterator *SPDataTypeListIteratorRef;
```

A reference to PICA's internal lists is accessed using a function in its runtime suite.

```
SPAPI SPErr (*GetRuntimeDataTypeList)( SPDataTypeListRef
*dataTypeList );
```

You might also generate your own list for the data type and use your stored reference to it. As a short cut to accessing PICAs internal lists, you can pass a NULL to function's requiring a list reference

Once a reference to the list to be traversed has been obtained, you will need a list iterator to traverse it. This is generated with the data type's NewListIterator( ) function. When it is no longer needed it will be disposed with data type's DeleteListIterator( ) function.:

```
SPAPI SPErr (*NewDataTypeListIterator)( SPDataTypeListRef
dataTypeList,
                        SPDataTypeListIteratorRef *iter );
SPAPI SPErr (*DeleteDataTypeListIterator)(SPDataTypeListIteratorRef
iter );
```

Once the list iterator is created, you can use the data type's Next( ) function to access each item in the list. The first time the Next( ) function is called, the first item in the list will be returned. Consecutive calls return the next item in the list. When the data type reference returned is NULL, the end of the list has been reached.

```
SPAPI SPErr (*NextDataType)( SPDataTypeListIteratorRef iter,
                        SPDataTypeRef *dataType );
```

There is no way to return to the first item in the list, except to destory the iterator and create a new one. Some lists have a Find( ) function which can be used as a more direct way of accessing known entries. Lists are not traversed often; if a data type reference is frequently needed it is simply stored by the plug-in.

```
SPAPI SPErr (*FindDataType)( SPDataTypeListRef dataTypeList,
                            char *name, SPDataTypeRef *dataType );
```

Once the data type reference has been retrieved, the data type's suite function can be used to access it if necessary

To add a new entry to a list, the data type's Add( ) function is used. This will take as arguments the list reference to which it is to be added and whatever information is needed to add a valid data object.

```
SPAPI SPErr (*AddDataType)( SPDataTypeListRef dataTypeList, void
*data );
```

Plug-ins can create data type lists of their own if needed, except for the files list. An example might be if an adapter wants to keep its plug-ins private rather than adding them to the main plug-in list. There are two functions for doing this.

```
SPAPI SPErr (*AllocateDataTypeList)( SPDataTypeListRef *dataTypeList
);
                        // may also take: SPStringPoolRef strings
SPAPI SPErr (*FreeDataTypeList)( SPDataTypeListRef dataTypeList );
```

The data type's Allocate( ) function creates the list. Many Allocate( ) functions take a string pool reference as an adational parameter. You can use the runtime suite to get PICA's pool and use it for the list's strings. If you create your own pool for the data type list you would pass it's reference. When the list is no longer needed, the data type's Free( ) function is used to dispose of it. If you have created a string pool for the list, you will alsoneed to dispose it. The list must be disposed when the application quits.

Error codes

PICA uses four-byte values for error codes. Often these are mnemonic 4 byte character codes, for instance Adobe Illustrator used 'STOP' for kCanceled-Error. Two universal errors defined by PICA are `kSPNoError` and `kSPUnimplemented-Error`.

```
#define kSPNoError            0x00000000L
#define kSPUnimplementedError '!IMP'
```

Other error codes should be defined by the returning suite and are defined in its header file.

5 The SPAccess Suite

About the SPAccess Suite

PICA's access suite is used to load and unload plug-ins. Its functions are used by those of the PICA interface suite in a simpler manner. When a plug-in is loaded, a plug-in access is created. The access suite provides other functions to obtain information about plug-in accesses.

Accessing the Suite

The SPAccess suite is referred to as:

```
#define kSPAccessSuite "SP Access Suite"
```

with the version constant:

```
#define kSPaccessSuiteVersion 2
```

It is acquired with the SPBasic suite as follows:

```
SPAccessSuite *sSPAccess;
error = sSPBasic->AcquireSuite( kSPAccessSuite,
                                kSPaccessSuiteVersion, &sSPAccess );
if ( error ) goto error;
```

Calling Other Plug-ins

The SPInterface suite provides a convenient way to send messages to other plug-ins. It actually uses the SPAccess suite to send the message. The SPAccess suite functions cab be used directly to achieve a similar result.

The code to call a plug-in using the access suite would look something like this:

```
SPErr SendMessage( SPPluginRef plugin, char *caller, char *selector,
                                        void *message, SPErr *error ) {
    SPErr result;
    SPAccessRef access;

    error = sAccess->AcquirePlugin( plugin, &access );
    if ( error ) goto error;

    error = sAccess->CallPlugin(access, caller, selector, message,
result);
    if ( error ) goto error;

    error = sAccess->ReleasePlugin( access );

error:
    return result;
}
```

The plug-in is first aquired causing PICA to load it into memory if necessary. The CallPlugin( )function will pass the caller, selector, and message to the plug-in's entry point. When the plug-in call returns, you release it.

SPAccess Information

An `SPAccessRef` is returned by the `AcquirePlugin( )` function. It is an access path to an open Sweet Pea plug-in. When you acquire a plug-in it will be loaded into memory if necessary. If it is already loaded, its existing access reference is returned. When you are done using the SPAccessRef, you release it using `ReleaseePlugin( )`. The SPAccessRef is used primarily for making calls to a plug-in.

SPAccessRef are reference-counted, meaning the number of times the plug-in has been acquired is kept. While the access count is non-zero, the plug-in is in use and will not be unloaded from memory. Releasing a plug-in decrements its access count. If its count is zero, the plug-in is unused and it can be unloaded from memory if needed. This means it's very important to balance all of your calls to `AcquirePlugin( )` and `ReleasePlugin( )`.

An `SPPlatformAccessInfo` structure contains platform-specific information about the access path to an open plug-in.

```
typedef struct {
    void *TopMapHndl;
    short CurMap;
} SPMacResChain;

#ifdef MAC_ENV
typedef struct {
    SPMacResChain *resources;
    unsigned long lastAccessTicks;
} SPPlatformAccessInfo;
#endif

#ifdef WIN_ENV
typedef struct {
    void *library;
    unsigned long lastAccessTicks;
} SPPlatformAccessInfo;
#endif
```

On the Mac it contains resource chain information and a time stamp of when it was last called. On Windows it contains the handle to the plug-in's library and the time stamp.

The access info is used by plug-ins that export suites. They must manually establish a resource context within any of their suite procedures before they can access their resources. This is not necessary when a plug-in is called through its entry point, since `CallPlugin( )` sets up the resource context for you. If a plug-in suite function failed to set up a resource context, the resources of the calling plug-in would be active.

SPAccess Suite Functions

AcquirePlugin( )

Loads a plug-in into memory so that it can be called

```
SPAPI SPErr AcquirePlugin( SPPluginRef plugin, SPAccessRef *access );
```

Given a reference to a Sweet Pea plug-in, this function loads the plug-in file if necessary and prepares it to be called. It returns an *access* reference, which you must use when calling the plug-in and which you must give to `ReleasePlugin`() when you are through.

Note: You can acquire yourself if you want to stay loaded in memory even when you're not being reference by anyone.

ReleasePlugin( )

Allows a plug-in to be unloaded from memory

```
SPAPI SPErr ReleasePlugin( SPAccessRef access );
```

You must call ReleasePlugin() when you are through with a plug-in that you had previously acquired. ReleasePlugin() adjusts the reference count of the plug-in, possibly making it eligible for unloading. After releasing a plug-in the access reference is no longer valid.

GetPluginAccess( )

Gets the current access reference of a plug-in

```
SPAPI SPErr GetPluginAccess( SPPluginRef plugin, SPAccessRef *access
            );
```

This function returns the current *access* of a plug-in *plugin*. If the plug-in is loaded, its SPAccessRef will be returned. If the plug-in is not loaded, the function will return NULL in access.

GetAccessPlugin( )

Gets the plug-in of an access reference

```
SPAPI SPErr GetAccessPlugin( SPAccessRef access, SPPluginRef *plugin
            );
```

Given an access reference, this function returns the plug-in to which it belongs.

GetAccessEntry( )

Get the entry point of an access

```
SPAPI SPErr GetAccessEntry( SPAccessRef access, SPEntry *entry );
```

The function returns the entry point of a plug-in access. When the Call-Plugin( ) function is used, it jumps to this location The entry point of the plug-in is defined as:

```
typedef SPAPI SPErr (*SPEntry)( char *caller, char *selector,
                                                    void *message );
```

The C string arguments *caller* and *selector* tell why the plug-in is being called, and the data structure pointed to by *message* contains any data associated with it.

Yet another way to call a plug-in would be to acquire it, get the entry of the returned access, and call the entry directly. Before calling the entry point, you would need to make the plug-in the current one with `SetCurrentPlugin()`.

GetAccessCount( )

Get the number of times a plug-in has been acquired

```
SPAPI SPErr GetAccessCount( SPAccessRef access, long *count );
```

The number of times the plug-in has been acquired is kept by PICA. The *count* for a given *access* is returned by this function.

While the access count is non-zero, the plug-in is in use and will not be unloaded from memory. Releasing a plug-in decrements its access count. If its count is zero, the plug-in is unused and it may be unloaded from memory if needed.

GetAccessInfo( )

Get the access information for an access reference

```
SPAPI SPErr GetAccessInfo( SPAccessRef access,
            SPPlatformAccessInfo *info );
```

An `SPPlatformAccessInfo` structure contains platform-specific information about the access path of an open plug-in. This function returns that a pointer to that *info* for the specified *access* reference.

The access info is used by plug-ins that export suites.

CallPlugin( )

Calls a plug-in with the specified message

```
SPAPI SPErr CallPlugin( SPAccessRef access, char *caller, char *se-
            lector, void *message, SPErr *result );
```

Calls the plug-in referenced by *access*, sending it the *caller*, *selector*, and *message* data pointers. The return value of the plug-in is returned to the caller in *result*.

GetCurrentPlugin( )

Gets the current plug-in

```
SPAPI SPErr GetCurrentPlugin( SPPluginRef *plugin );
```

This function returns the current plug-in.

The access information of the current plug-in is the current resource context.

It is the one last specified by `SetCurrentPlugin( )`. In many cases this is the last plug-in called by PICA's `CallPlugin( )` function. It may be a plug-in specified by a plug-in adapter.

The current plug-in is not necessarily the one currently running because of plug-in suite functions and other callback functions. For instance, if one plug-in calls a suite function provided by a second plug-in, the first plug-in remains the current plug-in while the function of the second is being executed.

SetCurrentPlugin( )

Sets the current plug-in

```
SPAPI SPErr SetCurrentPlugin( SPPluginRef plugin );
```

Sets the indicated plug-in to be PICA's current plug-in, making its resource context current.

An API adapter that keeps its plug-in references in PICA's plug-in list can use this function to set up a resource context before calling one of them. Before changing the current plug-in with this function, you should save the plug-in reference of the existing current one. When your plug-in no longer needs to be current, the previous state should be restored.

6 The SPAdapters Suite

About the SPAdapters Suite

PICA provides a set of services for managing plug-ins. The actual interfacing of the manager and any plug-in is done via an adapter. This includes actions such as telling the plug-in to do basic interfacing such as startup and shut-down, and do the only standard user interaction, show an about box. PICA plug-ins are controlled by an adapter that is built into the manager, so that the adapter architecture is used in all cases.

Adapters can be added by PICA plug-ins and used to support other plug-in APIs, providing a standard mechanism for handling backward compatiblity. When PICA has found and started up all of its plug-ins with its internal adapter, it will check the adapter list to see if any plug-in adapters were added. If so, it will call each one and give it an opportunity to start up the plug-ins it supports.

The adapter's job is to translate the legacy plug-in's API calls into those supported by PICA and host application, including user interface items and data translation. In addition to handling basic interface tasks, an adapter needs to provide a suite or suites of functions that allow the host and other plug-ins to call its plug-ins. The SPInterface suite is an example of this.

Accessing the Suite

The SPAdapters suite is referred to as:

```
#define kSPAdaptersSuite "SP Adapters Suite"
```

with the version constant:

```
#define kSPAdaptersSuiteVersion 3
```

It is acquired with the SPBasic suite as follows:

```
SPAdaptersSuite *sSPAdapters;
error = sSPBasic->AcquireSuite( kSPAdaptersSuite,
                                kSPAdaptersSuiteVersion, &sSPAdapters
);
if ( error ) goto error;
```

Plug-in Adapters

A plug-in making an adapter available does so when it receives the inter-face/startup message. The plug-in obtains the adapter list from PICA and uses the SPAdapter suite's function AddAdapter( ), like this:

```
SPAdapterRef       oldAPI;
SPAdapterListRef   adapterList;
SPErr error;

error = sSPRuntime->GetRuntimeAdapterList( &adapterList);
error = sSPAdapters->AddAdapter( adapterList, message->d.self,
                                    “old API adapter”, &oldAPI );
```

Adapter Messages

Once added, the plug-in will be called with the caller `kSPAdaptersCaller` and a number selectors and is expected to respond to them.

```
#define kSPAdaptersCaller             "SP Adapters"
#define kSPAdaptersStartupSelector    "Start up"
#define kSPAdaptersShutdownSelector   "Shut down"
#define kSPAdaptersFlushSelector      "Flush"
#define kSPAdaptersDisposeInfoSelector"Dispose info"
#define kSPAdaptersAboutSelector      "About"
#define kSPAdaptersFindPropertySelector"Find property"

#define kSPAdaptersAcquireSuiteHostSelector"Acquire Suite"
#define kSPAdaptersReleaseSuiteHostSelector"Release Suite"
```

The first of the selectors, `kSPAdaptersStartupSelector`, is a notification to startup. When this is received, PICA will have completed its plug-in start up process. The adapter can use the PICA file list as a source for potential plug-ins and load and issue appropriate startup commands to its plug-ins. Note that this is the second start up selector the a plug-in will receive, though from a different caller. The adapter `kSPAdaptersShutdownSelector` message action will be received when the application is about to quit running and allows the adapter to notify its plug-ins of this event. The adapted plug-in would do any necessary cleanup at this time, such as releasing system resources. The adapter plug-in will receive a normal interface shut down selector to do its own shut down process. To provide feedback to the user about its startup and shutdown process, the adapter should inform the host about each of its plug-ins using the `startupNotify( )` and `shutdownNotify( )` host functions available through the SPRuntime suite.

The flush selector, `kSPAdaptersFlushSelector`, indicates that the host application is trying to free memory and the adapter should do garbage collection, unloading an unused plug-ins. The dispose selector, `kSPAdaptersDisposeInfoSelector`, is akin to a final flush message. It is received after the adapter has shut down its plug-ins. Any memory occupied by its plug-ins should be freed, as should any memory used to keep track of them, including private PICA lists or string pools. This action is triggered by a called to the SPCaches suite function, `SPFlushCaches( )`.

The about selector, `kSPAdaptersAboutSelector`, indicates that the host application wants the plug-in indicated by the message structure to displayinformation about itself. This is equivalent to the about selector sent to a PICA plug-in.

The remaining message actions sent to a plug-in adapter are used if it can adapt a foreign function exporting system to PICA’s suite mechanism. In this case, the adapter would be expected to provide properties that are usable by PICA in finding the exported suites, and to load and unload the plug-in providing the function tables. The adapter should return a PiPL for one of its plug-ins when the `kSPAdaptersFindPropertySelector` selector is received. Whether the adapter makes a PiPL or has the plug-in supply it is up to you.

When an acquired suite is provided by an adapted plug-in not in memory, the plug-in's adapter will be called to load it into memory. The adapter will receive the selector `kSPAdaptersAcquireSuiteHostSelector`. When the suite is no longer needed, its providing plug-in's adapter will receive a release notification, `kSPAdaptersReleaseSuiteHostSelector`. The adapter can unload the plug-in at this time or wait for flush message.

Message Data

The message data structure received with a plug-in message is:

```
typedef struct SPAdaptersMessage {

    SPMessageData d;

    SPAdapterRef adapter;

    struct SPPlugin *targetPlugin;
    SPErr targetResult;

    /* for Find property selector */
    PIType vendorID;
    PIType propertyKey;
    long propertyID;
    void *property;    /* returned here */

    /* for Flush selector */
    SPFlushCachesProc flushProc;
    long flushed;      /* returned here */

    /* for Suites selectors */
    struct SPSuiteList *suiteList;/* use these if you need name,
    struct SPSuite *suite;     apiVersion, or internalVersion info */
    struct SPPlugin *host;     /* plug-in hosting the suite, to be
                               aquired/released by adapter */
    void *suiteProcs;  /* returned here if reallocated */
    long acquired;     /* returned here */

} SPAdaptersMessage;
```

The SPMessageData structure and the adapter reference will always be valid when the plug-in is called. The adapter reference is that returned by the AddAdapter( ) function and is used to indicate the adapter to which the message is being sent. If a plug-in adds more than one adapter it will need this information to know how to handle the message. Each of the other message selectors will be passed necessary information in the message data structure.

kSPAdaptersStartupSelector
kSPAdaptersShutdownSelector
kSPAdaptersDisposeInfoSelector
The startup, shutdown, and dispose info adapter selectors receive no other information in the structure.

kSPAdaptersAboutSelector
If the adapter is called to display information about a particular plug-in, the message structure member *targetPlugin* will indicate the plug-in. The adapter should send the targetPlugin the correct about message. If the about message returns a result, it can be returned to the host in the *targetResult* field.

kSPAdaptersFlushSelector

If the message selector is an adapter flush action, it will receive a function pointer in *flushProc*. This function is likely supplied by the host application and is called by the adapter to determine which plug-ins are removed from memory. The number of plug-ins the adapter flushes from memory should be returned in *flushed*. The flush function prototype is:

```
typedef SPErr (*SPFlushCachesProc)(char *type, void *data, long
*flushed);
```

The *type* and *data* are set by the calling adapter. The type is a C-style string which indicates the type of the plug-in and the data structure contents to the flush procedure. If it is to be removed, true is returned in *flushed*. The adapter actually unloads the plug-in from memory.

```
#define kMyPluginAdapterFlushType "My Plugin Adapter"
typedef struct {
    long lastAccess;
    long whateverElse;
} MyFlushData

MyFlushData flushData;
long flush, flushCount = 0;

for ( i = 0; i < kMyPluginCount; i++ ) {
    flushData.lastAccess = myPlugins[ i ].lastAccess;
    message->flushProc(kMyPluginAdapterFlushType, &flushData,
&flush);
    if ( (!myPlugins[ i ].inUse ) && flush ) {
        unloadPlugin( myPlugins[ i ] );
        flushCount++;
    }
}

message->flushed = flushCount;
```

The adapter should provide information in a header file with the type string used to identify it and the data structure it will pass to the flush function. The host application supporting the adapter would use this information within its flush function handle whether a plug-in of the type should be unloaded. If the host is unfamiliar with the plug-in type, it may return a default value of true or false.

```
SPErr HostFlushCachesProc(char *type, void *data, long *flushed) {
    if ( strcmp( type, kMyPluginAdapterFlushType ) == 0 ) {
        if ( now - (MyFlushData*)data->lastAccess < kTwoMinutes )
            *flushed = true;

    } else if ( strcmp( type, kOtherPluginAdapterFlushType ) == 0 ) {
        ...

    } else
        *flushed = true;
}
```

If a plug-in wants to flush plug-ins from memory, it can use the SPCaches suite to call all adapters with a `SPFlushCachesProc( )` function. Its flush function would need to respect the types and data structures of other adapters.

kSPAdaptersFindPropertySelector
PICA keeps a list of properties with each plug-in in its list. If an adapter adds plug-ins to PICA's main plug-in list, they will be search for exports. The find property selector will be sent to the adapter to obtain property information about plug-ins it added.

The plug-in for which PICA needs a property is indicated by the *targetPlugin* field of the message. The needed property is indicated by the three property identification fields:

```
/* for Suites selectors */
PIType vendorID;
PIType propertyKey;
long propertyID;
```

These are as described in the PiPL chapter. The adapter should generate, load, or otherwise obtain the property information for the plug-in. The property data is returned in the structure member:

```
void *property;    /* returned here */
```

The pointer should be to a valid property. If the property requested is unavailable, the plug-in can return NULL for the pointer.

kSPAdaptersAcquireSuiteHostSelector
kSPAdaptersReleaseSuiteHostSelector
The last two selectors need to be handled only if the adapter allows plug-ins to provide function suites that are stored in PICA's main suite list and identified by export properties ('expt'). If PICA needs such a suite and the plug-in providing it is currently unavailable, the plug-in's adapter will be called with an acquire suite host selector. The information about the suite to be acquire is in the message data structure:

```
/* for Suites selectors */
struct SPSuiteList *suiteList;
struct SPSuite *suite;
struct SPPlugin *host;    /* plug-in hosting the suite, to be
                          aquired/released by adapter */
void *suiteProcs;  /* returned here if reallocated */
long acquired;     /* returned here */
```

Given the *suiteList* and *suite* fields, you can obtain the name, version, and other information about the suite to be acquired. The plug-in that provides the suite and that must be acquired is indicated in the *host* field. It may be that knowing the *host* plug-in provides enough information about the suite for it to be acquired, in which case the *suite* and *suiteList* fields are unnecessary. If reloading the plug-in changes the suite's location in memory, a pointer to the reallocated suite can be returned in the *suiteProcs* pointer. If the loading process merely restores the function pointers in an existing memory block, NULL may be returned for the *suiteProcs* pointer. If the suite acquire is successful, *acquired* should be set to true; *acquired* should be set to false if for some reason acquiring the suite host fails.

When a suite made available through an adapter has been released by all plug-ins, the adapter will receive the release suite host selector. This indicates that the adapter should unload the plug-in providing the suite from memory. The information about the released suite is in the same fields as the adapter acquire suite information.

SPAdapters Suite Functions

AllocateAdapterList( ) — Create a new adapter list

```
SPAPI SPErr AllocateAdapterList( struct SPStringPool *stringPool, SP-
            AdapterListRef *adapterList );
```

Allocates a new list of SP adapters, returned in adapterList. Adapter names added to the new adapter list will be stored in the specified stringPool.

You can also use PICA's main adapter list, available through the SPRuntime suite. See the discussion on PICA lists in the chapter, "The PICA Core."

FreeAdapterList( ) — Dispose of an adapter list

```
SPAPI SPErr FreeAdapterList( SPAdapterListRef adapterList );
```

Disposes of the specified adapter list and any entries in it. The adapter list was created with the AllocatedAdapterList( ) function.

See the discussion on PICA lists in the chapter, "The PICA Core."

AddAdapter( ) — Add a new an adapter to a list

```
SPAPI SPErr AddAdapter( SPAdapterListRef adapterList,
            struct SPPlugin *host, char *name, long version,
            SPAdapterRef *adapter );
```

Adds an adapter to the indictated *adapterList*. Pass NULL to use PICA's main adapter list. The *host* is a reference to the plug-in adding the adapter. The *name* is a C string identifying the adapter. Plug-ins added by the adapter will use this string to identify their host. Only the latest *version* of an adapter will be allowed to startup plug-ins. A reference to the added *adapter* will be returned. If a plug-in adds more than one adapter, it should save this value and use it to determine which adapter is being sent a message. If the plug-in only adds one adapter, you can pass NULL for adapter reference.

```
sSPAdapters->AddAdapter( nil, message->d.self,
                  "MYAPP Legacy Plug-in Adapter", 1, NULL );
```

SPFindAdapter( ) — Find an adapter by name

```
SPAPI SPErr SPFindAdapter( SPAdapterListRef adapterList, char *name,
            SPAdapterRef *adapter );
```

Get a reference to the adapter identified by *name*.

See the discussion on PICA lists in the chapter, "The PICA Core."

NewAdapterListIterator( ) — Create an iterator to traverse a list

```
SPAPI SPErr NewAdapterListIterator( SPAdapterListRef adapterList,
            SPAdapterListIteratorRef *iter );
```

Returns an iterator to access an adapter list. The iterator is set to the first adapter in the list.

See the discussion on PICA lists in the chapter, "The PICA Core."

NextAdapter( ) — Advance to the next entry in a list

```
SPAPI SPErr NextAdapter( SPAdapterListIteratorRef iter,
            SPAdapterRef *adapter );
```

Advances the list iterator to the next adapter in the list. When the last adapter has been reached, this function will return NULL.

See the discussion on PICA lists in the chapter, "The PICA Core."

DeleteAdapterListIterator( ) — Dispose of an adapter list iterator

```
SPAPI SPErr DeleteAdapterListIterator( SPAdapterListIteratorRef iter
            );
```

When a list iterator is no longer needed it should be disposed.

See the discussion on PICA lists in the chapter, "The PICA Core."

GetAdapterHost( ) — Get the plug-in hosting an adapter

```
SPAPI SPErr GetAdapterHost( SPAdapterRef adapter,
            struct SPPlugin **plugin );
```

Returns a reference to the plug-in that added adapter in *plugin*. You might want this reference to send the adapter a message.

GetAdapterName( ) — Get the identifier string of an adapter

```
SPAPI SPErr GetAdapterName( SPAdapterRef adapter, char **name );
```

Get a pointer to the name of an adapter, which is used to identify it. The string should not be modified.

To determine how to call a plug-in you need to first identify its adapter. For instance, plug-ins added by PICA's built-in adapter have the identification string:

```
#define kSPSweetPea2Adapter"Sweet Pea 2 Adapter"
```

To verify that a message could be sent to a plug-in, you would do this:

```
SPErr error;
SPPluginRef pluginToCall;
SPAdapterRef pluginsAdapter;
char *adapterName;
long adapterVersion;

error = sSPPlugins->GetPluginAdapter( pluginToCall, &pluginsAdapter
);
error = sSPAdapters->GetAdapterName( pluginsAdapter, &adapterName );

if ( strcmp( adapterName, kSPSweetPea2Adapter ) == 0 ) {
    // it is a PICA plug-in, call it as such with sSPInterface.
} else if ( strcmp( adapterName, "MYAPP Legacy Plug-in Adapter" ) ==
0 ) {
    // it is an adapted plug-in, call it with the adapter's
    // interface suite
    error = sSPAdapters->GetAdapterVersion( pluginsAdapter,
                                            &adapterVersion );
    if ( adapterVersion == 1 ) {
        // use one hypothetical interface suite
    } else if ( adapterVersion == 2) {
        // use another hypothetical interface suite
    }
}
```

GetAdapterVersion( )

Get the version of an adapter

```
SPAPI SPErr GetAdapterVersion( SPAdapterRef adapter, long *version );
```

Get the version of an adapter

To determine how to call a plug-in you need to first identify its adapter. You may need to further identify it by its version. For instance, plug-ins added by PICA's built-in adapter currently have the version:

```
#define kSPSweetPea2AdapterVersion1
```

See the code example in `GetAdapterName( )`.

7 The SPBasic Suite

About the SPBasic Suite

API functions provided by PICA and host applications are organized into related groups called **suites**. The SPBasic suite is how plug-ins get to the rest of the suite universe. It is the bootstraps by which a plug-in pulls itself up: plug-ins use it to acquire and release all the other suites that they need.

A suite is a structure filled with function pointers. Before any API function can be used, its suite must be acquired using a function in the SPBasic Suite. Acquiring a suite returns a pointer to a structure filled with valid function pointers. When a suite is no longer needed it should be released, also done with a function in the basic suite.

The SPBasic Suite also provides a function to indicate that a suite function is no longer valid. Plug-ins adding suites will use that function when they are unloaded.

Accessing the Suite

The SPBasic suite is a part of plug-in data passed in the message data passed to a plug-in. It can be used from the message data structure like this:

```
message->d.basic->function( )
```

or, to make reading easier, assigned to a variable and used through it:

```
SPBasicSuite sBasic = message->d.basic;
sBasic->function( )
```

While there is really no need to acquire it, as with any suite there are definitions for doing so.

The basic suite is referred to as:

```
#define kSPBasicSuite     "SP Basic Suite"
```

with the version constant:

```
#define kSPBasicVersion   2
```

It is acquired with the basic suite (basically, using itself) as follows:

```
SPBasicSuite *sBasic2;
error = message->d.basic->AcquireSuite( kSPBasicSuite,
    kSPBasicVersion, &sBasic2 );
if ( error ) goto error;
```

or

```
error = sBasic->AcquireSuite( kSPBasicSuite, kSPBasicVersion,
&sBasic2 );
```

```
if ( error ) goto error;
```

What Is a Suite?

A function suite is simply a list of function pointers defined in a standard structure:

```
typedef struct {
    SPAPI AIFixed (*FixedRnd) ( void );
    SPAPI void  (*SetRndSeed) ( long seed );
    SPAPI long  (*GetRndSeed) ( void );
} AIRandomSuite;
```

Either the host application or a plug-in creates this table by setting the function pointers to valid routine addresses. This happens when the suite is acquired. When a suite provide by a plug-in is unavailable, the function pointer will set to the SPBasic suite function, `Undefined( )`.

Acquiring and Releasing Suites

Before the functions of a suite can be used the suite must be *acquired*. When the suite is no longer needed it is *released*. Suites are acquired when the plug-in has determined the selector type and released when the appropriate action has been taken. The SPBasic suite provides the functions to acquire and release suites.

Every suite is identified by a suite name (a C string) and version (integer) used to acquire a suite. A suite may exist in several versions with different functionality, so a plug-in must acquire the appropriate one. These identifiers are found in the header file for the suite, e.g. “AIRandom.h” for the random suite above.

To acquire a suite, a plug-in uses the `AcquireSuite( )` function of the SPBasic Suite and the identifiers of the desired suite. Since a suite is a function pointer filled structure, a pointer to that structure must be declared. A pointer to the acquired suite structure is returned in that pointer:

```
SPErr error;
SPBasicSuite *sBasic = message->d.basic;
AIRandomSuite *sRandom;

sBasic->AcquireSuite( kAIRandomSuite, kAIRandomVersion, &sRandom );
```

When finished, the suite is released using the suite identifiers:

```
sBasic->ReleaseSuite( kAIRandomSuite, kAIRandomVersion );
```

Using a Suite Function

The C language allows a program to call a function from a variable, essentially jumping to a memory location defined at runtime. To call a function in a suite, use the following syntax:

```
someNumber = sSomeSuite->SomeSuiteFunction( );
```

What this is doing is finding the structure member `SomeSuiteFunction( )`, getting a function address at that location, setting up the stack appropriately, and jumping to the function. Thanks to the C language, all this is hidden when actually writing your plug-in code.

Attempting to use a function from a suite that has not been acquired may cause your plug-in to crash.

When Suites Should Be Acquired

Do not acquire or release suites when PICA’s access (reload and unload) and property messages are received. This is because suites may be unavailable at

these times. The exception to this is that PICA's built suites will be available and can be used. The SPBlock suite, for instance, can be used to allocate memory for an SPProperties request.

The sample code accompanying this document provides a routine that acquires and releases all needed suites en mass, rather than individually. This means that suites are at times acquired even if they aren't going to be used. As long as they are released, this is usually acceptable.

At the plug-in shutdown messsage, you should avoid doing this mass acquire and acquire only the suites needed to shut down the plug-in. For instance, a plug-in might free memory and save preferences, requiring the blocks and preferences suites. There is no need to acquire a raster access suite at this time. The reason for treading lightly at shutdown is to avoid acquiring suites provided by plug-ins that have already been shutdown. Such an action will be successful (the plug-in is restarted), but is certainly less efficient than it needs to be.

SPBasic Suite Functions

AcquireSuite( )

Acquire a suite of functions

```
SPAPI SPErr AcquireSuite(
            char *name,
            long version,
            void **suiteProcs );
```

Acquiring a suite returns a pointer to a valid set of function addresses in *suiteProcs*. The identifiers of the suite are passed in the *name* argument and the *version* argument.

The name and version identifiers for a suite are found in its associated header file. When a suite is acquired through the SPBasic suite, the latest available version will be supplied.

The following code sample shows how to acquire the random number suite:

```
AIRandomSuite *sRandom;
error = message->d.basic->AcquireSuite( kAIRandomSuite,
    kAIRandomVersion, &sRandom );
if ( error ) goto error;
```

See also: ReleaseSuite( )
The SPSuites Suite chapter

ReleaseSuite( )

Release a function suite that is no longer needed.

```
SPAPI SPErr ReleaseSuite (
            char *name,
            long version );
```

For efficiency, PICA keeps track of which suites are currently being used. When you are done using a suite, you should release it. This allow PICA to unload suite providing plug-ins if no one is using their suites.

The suite identifiers are passed to `ReleaseSuite( )`. The error returned by the function can be ignored.

```
message->d.basic->ReleaseSuite( kAIRandomSuite, kAIRandomVersion );
```

See also: AcquireSuite( )
The SPSuites Suite chapter

Undefined( )

Assinged to function pointers of a suite when its providing plug-in is unloaded.

```
SPAPI SPErr Undefined (
            void );
```

When a plug-in providing a suite is unloaded from memory, its suite pointers become invalid. They should be set to this function pointer to indicate this.

When the plug-in is reloaded the suite functions can be reassigned to valid functions.

```
UnloadFunctions( SPBasic *sBasic ) {
    mySuite->function1 = sBasic->Undefined;
    mySuite->function2 = sBasic->Undefined;
    mySuite->function3 = sBasic->Undefined;
}
```

See also: The SPSuites Suite chapter

8 The SPBlocks Suite

About the SPBlocks Suite

The functions in this suite are used to request, release, and resize blocks of memory. They are analagous to the standard C library memory allocation routines or Macintosh Toolbox pointer routines. Using these calls will make porting a plug-in to different platforms easier. In some cases, the plug-in's host may expect memory allocated in a certain manner and the use of these calls may be required.

Accessing the Suite

The SPBlocks Suite is referred to as:

```
#define kSPBlocksSuite "SP Blocks Suite"
```

with the version constant:

```
#define kSPBlocksSuiteVersion 2
```

It is acquired with the Basic Suite as follows:

```
SPBlocksSuite *sSPBlocks;
error = sSPBasic->AcquireSuite( kSPBlocksSuite,
                          kSPBlocksSuiteVersion, &sSPBlocks);
if ( error ) goto error;
```

SPBlocks Suite Functions

AllocateBlock( )

Allocate a new block of memory

```
SPAPI SPErr AllocateBlock ( long size, char *debug, void **block );
```

This call is similar to the standard C library malloc( ) routine and the Macintosh Toolbox NewPtr( ). It allocates a block of *size* bytes and returns a pointer *block* to the beginning of the memory. If non-NULL, the C string pointed to by *debug* is used to tag the block if a development version of the PICA host application is available. Otherwise it is ignored.

The following example would create a block of memory for storing a block of text:

```
char *text;

error = sSPBlocks->AllocateBlock( kStartTextSize, &text );
if ( error ) goto error;
```

If the function fails due to lack of memory, it will return **`kSPOutOfMemoryErr`**.

See also: DisposeBlock( )
ReallocateBlock( )

FreeBlock( )

Release a block of memory allocated by the AllocateBlock( ) function

```
SPAPI SPErr FreeBlock( void *block );
```

FreeBlock( ) is the opposite of AllocateBlock( ) and is similar to standard C library free( ) function and the Macintosh Toolbox DisposePtr( ). The memory pointed to by *block* is returned to the application heap.

```
if ( text != NULL ) {
    sBlock->DisposeBlock( text );
    text  = NULL;
}
```

See also: AllocateBlock( )

ReallocateBlock( )

Change the size of a block previously allocated with AllocateBlock()

```
SPAPI SPErr ReallocateBlock( void *block, long newSize, void **new-
            Block );
```

The original memory pointer is passed in for *block*. The desired size for the block is specified by *newSize*. The reallocated memory pointer will be returned in the *newBlock* pointer.

This function is similar to realloc( ) and SetPtrSize( ). ReallocateBlock( ) will try to increase the size of the block without changing its location. If there is not room on the heap, the block will be moved and the new location returned in the *newBlock* argument.

```
    if (text != NULL) {
        error = sSPBlock->ReallocateBlock( text,
                        kStartTextSize + kTextBlockSize, &text);
        if ( error ) goto error;
    }
```

If the function fails due to lack of memory, it will return **`kOutOfMemoryErr`**.

See also: AllocateBlock( )
DisposeBlock( )

9 The SPCaches Suite

About the SPCaches Suite

PICA plug-ins are intended to move in and out of memory as necessary, allowing a small a memory footprint as desired. By default, PICA will keep in memory, or cache, loaded plug-ins until the host application heap has been filled and then unload them. The unloading of cached plug-ins from memory beyond this simple strategy is handled by the host using the single function in the SPCaches suite.

Accessing the Suite

The SPCaches suite is referred to as:

```
#define kSPCachesSuite "SP Caches Suite"
```

with the version constant:

```
#define kSPCachesSuiteVersion 2
```

It is acquired with the SPBasic suite as follows:

```
SPCachesSuite *sSPCaches;
error = sSPBasic->AcquireSuite( kSPCachesSuite,
                                kSPCachesSuiteVersion, &sSPCaches );
if ( error ) goto error;
```

SPAdapters Suite Functions

SPFlushCaches( ) Create a new adapter list

```
SPAPI SPErr SPFlushCaches( SPFlushCachesProc flushProc, long
*flushed );
```

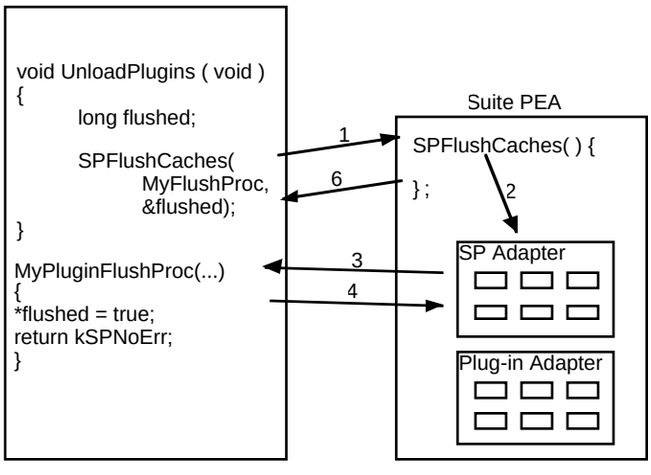
The single function in this suite causes a message to be sent to all plug-in adapters telling them to flush (unload) any unused plug-ins from memory. The function *flushProc* is used to determine which plug-ins to unload. The return value *flushed* indicates the number of plug-ins flushed from memory. It is unlikely that a plug-in will need to call this function.

Each adapter will call *flushProc* for any plug-ins that it can unload, returning control of the unload decision to the original caller of SPFlushCaches( ). The original caller returns whether or not a plug-in should be flushed. The flush function prototype is:

```
typedef SPErr (*SPFlushCachesProc)(char *type, void *data, long
*flushed);
```

The *type* and *data* are set by the adapter. The type is a C-style string which indicates the type of the plug-in and the data structure contents. The flush-Proc uses this information to determine whether the plug-in should be unloaded. If it is to be removed, true is returned in the flushProc's *flushed* argument. The adapter actually unloads the plug-in from memory.

Figure 1 show the flow of control for the flush process:



1. Someone calls SPFlushCaches( ) with a flushProc (Control is in the original caller of SPFlushCaches( ))

2. Suite PEA sends each of adapter a flush message that includes the flushProc.
(Control is in Suite PEA, wihin the host app)

3. An adapter puts information on one of its plug-ins in a data structure and calls the flushProc with this information and an identifier.
(Control is in the the adapter, which maybe a plug-in)

4. The flushProc uses the identifier to determine the structure contents. It uses the data to determine if the plug-in should be unloaded, returning true or false.
(Control is in the original caller)

5. The adapter unloads the plug-in if necessary.
The adapter repeats 3 for each of its plug-ins.
(Control is in the the adapter)

Repeat 2 for each adapter

6. Return the flush count to the original caller
(Control is returned to the original caller of SPFlushCaches( ))

Figure 1 - PICA's Plug-in Flush Process

The simplest possible flushProc would ignore the type and data contents and always return true. This would for a flush of all possible plut-ins.

```
SPErr FlushAllCachesProc(char *type, void *data, long *flushed) {
    *flushed = true;
}
```

The flushProc should respect the type and data of the available adapters. This information should be available in the header files used to build adapters.

10 The SPFiles Suite

About the SPFiles Suite

The SPFiles suite is used to access PICAs file list. This list is create at startup and contains references to every file in the plug-in folder of the host application. This includes resolved file and folder aliases if the host platform supports this feature.

PICA uses the file list to scan for its plug-ins. It maintains the file list and provides a suite to access it to avoid redundant directory scans. Adapters looking for their own plug-ins and PICA plug-ins looking for support files can scan the list to locate relevant files rather than walking platform directory structures on their own. By using it, they avoid repeatedly walking the directory hierarchy and ensure consistent behavior

Accessing the Suite

The SPFiles suite is referred to as:

```
#define kSPFilesSuite "SP Files Suite"
```

with the version constant:

```
#define kSPFilesSuiteVersion 3
```

It is acquired with the SPBasic suite as follows:

```
SPFilesSuite *sSPFiles;
error = sSPBasic->AcquireSuite( kSPFilesSuite,
                                kSPFilesSuiteVersion, &sSPFiles );
if ( error ) goto error;
```

Platform File Specifications

PICA uses a SPPlatformFileSpecification to reference a file. This structure is specific to the platform on which PICA is running. The Macintosh and Windows structures are:

```
#ifdef MAC_ENV
typedef struct {  /* identical to FSSpec */
    short vRefNum;
    long parID;
    unsigned char name[64];
} SPPlatformFileSpecification;
#endif

#ifdef WIN_ENV
typedef struct {
    char path[300];
} SPPlatformFileSpecification;
#endif
```

SPFiles Suite Functions

AllocateFileList( ) — Create a new file list

```
SPAPI SPErr AllocateFileList( SPFileListRef *fileList );
```

Allocates a new list of SP files, returned in *fileList*.

You can also use PICA's main file list, available through the SPRuntime suite. See the discussion on PICA lists in the chapter, "The Core."

FreeFileList( ) — Dispose of a file list

```
SPAPI SPErr FreeFileList( SPFileListRef fileList );
```

Disposes of the specified *fileList* and any entries in it. The file list was created with the AllocatedFileList( ) function.

See the discussion on PICA lists in the chapter, "The Core."

AddFiles( ) — Add a new file or files to a list

```
SPAPI SPErr AddFiles( SPFileListRef fileList,
            SPPlatformFileSpecification *file );
```

This function adds a file or files to the indicated file list. If *file* specifies an individual file, it will be added to the list. If *file* specifies a directory, the directory will be recursively searched and all files in it will be added to the file list.

NewFileListIterator( ) — Create an iterator to traverse a list

```
SPAPI SPErr NewFileListIterator( SPFileListRef fileList,
            SPFileListIteratorRef *iter );
```

Returns an iterator to access a file list. The iterator is set to the first file in the list.

See the discussion on PICA lists in the chapter, "The Core."

NextFile( ) — Advance to the next entry in a list

```
SPAPI SPErr NextFile( SPFileListIteratorRef iter,
            SPFileRef *file );
```

Advances the list iterator to the next file in the list. When the last file has been reached, this function will return NULL.

See the discussion on PICA lists in the chapter, "The Core."

DeleteFileListIterator( )

Dispose of a file list iterator

```
SPAPI SPErr DeleteFileListIterator( SPFileListIteratorRef iter );
```

When a list iterator is no longer needed it should be disposed.

See the discussion on PICA lists in the chapter, "The Core."

GetFileSpecification( )

Get the platform file specification of a file list entry

```
SPAPI SPErr GetFileSpecification( SPFileRef file,
            SPPlatformFileSpecification *fileSpec );
```

Fills in the caller supplied file specification pointed to by **fileSpec* for the indicated *file*.

GetFileInfo( )

Get platform information about a file list entry

```
SPAPI SPErr GetFileInfo( SPFileRef file, SPPlatformFileInfo *info );
```

Fills in the caller supplied file info structure *info* with platform file information on the file list entry *file*.

The attibutes for a file are platform dependent:

```
#ifdef MAC_ENV
typedef struct {
    unsigned long attributes;
    unsigned long creationDate;
    unsigned long finderType;
    unsigned long finderCreator;
    unsigned short finderFlags;
} SPPlatformFileInfo;
#endif

#ifdef WIN_ENV
typedef struct {
    unsigned long attributes;
    unsigned long lowCreationTime;
    unsigned long highCreationTime;
    char *extension;
} SPPlatformFileInfo;
#endif
```

The file's attributes are standard platform data. Information about the file attributes is available from the platform vendor.

11 The SPInterface Suite

About the SPInterface Suite

One of the services which PICA provides is the ability for plug-ins to call other plug-ins. One way to do this is to send the plug-in a message through its main entry point. The same functions that PICA and the host application use to do this are provided to plug-ins in the SPInterface suite.

The SPPlugin and SPAccess suites can be used in conjunction to achieve a similar result. The SPInterface suite provides a more convenient approach.

Accessing the Suite

The SPInterface suite is referred to as:

```
#define kSPInterfaceSuite "SP Interface Suite"
```

with the version constant:

```
#define kSPInterfaceSuiteVersion 2
```

It is acquired with the SPBasic suite as follows:

```
SPInterfaceSuite *sSPInterface;
error = sSPBasic->AcquireSuite( kSPInterfaceSuite,
                                kSPInterfaceSuiteVersion, &sSPInter-
face );
if ( error ) goto error;
```

Calling Other Plug-ins

The SPInterface suite has three functions used to call other plug-ins with a specified message. The “About Plug-ins” feature of Adobe Illustrator is written using these calls to send it’s plug-ins an about message.

```
SPAPI SPErr SendMessage( SPPluginRef plugin, char *caller,
                         char *selector, void *message, SPErr *result );
SPAPI SPErr SetupMessageData( SPPluginRef plugin, SPMessageData
*data );
SPAPI SPErr EmptyMessageData( SPPluginRef plugin, SPMessageData
*data );
```

The functions might be used on a specific plug-in whose functionality your plug-in would like to use or in conjunction with the plug-in list to call all plug-ins.

The setup and empty functions bracket the SendMessage( ) function. The setup function will fill in an SPMessageData structure with the basic suite, the globals pointer kept by PICA for the plug-in, and the plug-in’s SPPlug-inRef reference (its reference to itself). If the message you will be sending to a plug-in requires other information in its data structure, the calling plug-in is responsible for setting it up. The empty function will undo the actions of

the setup call: releasing the basic suite and storing the globals pointer in case it was changed.

SendMessage( ) will load the specified plug-in into memory if necessary and pass the message action and data structure to its entry point. To the called plug-in it is as if the main application had done this. Here is how the process works:

```
SPInterfaceMessage aboutMessage;
SPErr error = 0, result = 0;

error = sSPInterface->SetupPluginData( plugin, &aboutMessage.d );
if ( error ) goto error;

error = sSPInterface->CallPlugin( plugin, kSPInterfaceCaller,
               kSPInterfaceAboutSelector, &pluginMessage, &result );
if ( error ) goto error;

error = sSPInterface->EmptyPluginData( plugin, &aboutMessage.d );
if ( error ) goto error;
```

Before using the SPInterface functions to send a message, the calling plug-in should use the SPAdapter suite `GetAdapterName( )` call to determine that it is a PICA plug-in.

Calling Non-PICA Plug-ins

The SPInterface suite functions will fail if they are used on a non-PICA plug-in. To call such a plug-in, you would instead use an analagous interface suite provided by the plug-in adapter hosting it. For instance, the Adobe Illustrator application uses a suite provided by an adapter plug-in to send messages to plug-ins written for Illustrator 5.5:

```
typedef struct {

    AIAPI AIErr (*SetupAIEffect50)( SPPluginRef plugin, AIEffect50
*pb );
    AIAPI AIErr (*EmptyAIEffect50)( SPPluginRef plugin, AIEffect50
*pb );

    AIAPI AIErr (*SetupAIPluginPB55)(SPPluginRef plugin,
AIPluginPB55 *pb);
    AIAPI AIErr (*EmptyAIPluginPB55)(SPPluginRef plugin,
AIPluginPB55 *pb);

    AIAPI long (*CallPlugin50)( SPPluginRef plugin, AIEffect50 *pb );
    AIAPI long (*CallPlugin55)( SPPluginRef plugin, AIPluginPB55 *pb
);

} AIBackwardSuite;
```

In this case, there is a similarity between the AIBackward and SPInterface suites. This may or may not be true. Check the information provided with the adapter's suite for how to access it's plug-ins.

See the SPAdapters suite chapter, `GetAdapterName( )` function to see how to determine the adapter supporting a plug-in.

SPInterface Suite Functions

SendMessage( )

Send a plug-in a message

```
SPAPI SPErr SendMessage( SPPluginRef plugin, char *caller, char *se-
            lector, void *message, SPErr *result );
```

Calls the specified plug-in with the given message action (*caller* and *selector*) and message data structure, *message*. The value returned by the called plug-in is in *result*.

SendMessage( ) will load the specified plug-in if necessary and pass the caller and selector and message data to its main entry point. To the called plug-in it is as if the main application had done this.

The setup and empty plugin data functions bracket the `SendMessage`( ) function. See the example in the Calling Other Plug-ins section above.

See also: SetupMessageData( )
EmptyMessageData( )

SetupMessageData( )

Prepares an SPMessageData structure for SendMessage( )

```
SPAPI SPErr SetupMessageData( SPPluginRef plugin, SPMessageData *da-
            ta );
```

The setup function will fill in an SPMessageData structure with the basic suite, the globals pointer kept by the PICA for the plug-in, and the plug-in's SPPluginRef reference (a reference to itself).

If the message which you will be sending a plug-in requires other information in the message data structure, you are responsible for setting it.

See also: EmptyMessageData( )
SendMesssage( )

EmptyMessageData( )

Clean up an SPMessageData structure after SendMessage( )

```
SPAPI SPErr EmptyMessageData( SPPluginRef plugin, SPMessageData *da-
            ta );
```

The empty function will undo the actions of the setup message function.

The basic suite is released and the globals pointer is stored in case it was changed.

See also: SetPluginGlobals( )
SetupMessageData( )
SendMesssage( )

StartupExport( ) — Start a plug-in exporting a suite

```
SPAPI SPErr StartupExport( SPPluginListRef pluginList, char *name,
            long version,long *started );
```

This function handles a special case of sending a plug-in a message: requesting that the plug-in exporting a suite be sent the startup message. *name* and *version* indicate the desired suite. Plug-ins in the indicated *pluginList* will be scanned for the appropriate export property. If found, the plug-in will exporting the suite will be started and true returned in the argument *started*. If not found, false will be returned in *started*.

12 The SPPlugins Suite

About the SPPlugins Suite

Functions in this suite are used to access the plug-ins of which the Suite PEA is aware. This includes its own plug-ins and the plug-ins added by API adapters. One plug-in could use the functions to determine the host adapter of another plug-in so that it could send the second plug-in a message. An adapter could use the functions to store references to its plug-ins.

Accessing the Suite

The SPPlugins suite is referred to as:

```
#define kSPPluginsSuite  "SP Plugins Suite"
```

with the version constant:

```
#define kSPPluginsSuiteVersion  2
```

It is acquired with the basic suite as follows:

```
SPPluginsSuite*    sSPPlugins;

// Use the Basic Suite as describe in the chapter of that name
// and to acquire the suite
error = sBasic->AcquireSuite( kSPPluginsSuite,
            kSPPluginsSuiteVersion, &sSPPlugins );
if ( error ) goto error;
```

Plug-in States

Suite PEA plug-ins have several states that can be inspected and set using the plug-in suite. These states will most likely be of interest to API adapters supporting their own plug-ins. The two states are *started* and *broken*.

The plug-in broken state indicates whether a plug-in is available to receive messages. If it is “broken” some error has occurred indicating it has become unavailable. An example of what might cause a plug-in to become broken is moving a plug-in file from the plug-in folder after it has started up.

The started state indicates whether a plug-in has received the interface startup message. Once a plug-in has returned after receiving the message, its started state will be set to true.

Host Plug-ins

Some "plug-ins" that ship with the host application are actually built into the application; these are called *host plug-ins*. A host might use a plug-in to add its suites to Suite PEA. This state can be determined by checking whether the plug-in has a host entry point. If the host entry point is non-NULL, the plug-in is a part of the application. If the host entry point is NULL, the plug-in is an external file.

SPPlugins Suite Functions

AllocatePluginList( ) — Create a new plug-in list

```
SPAPI SPErr AllocatePluginList( struct SPStringPool *stringPool,
            SPPluginListRef *pluginList );
```

Allocates a new list of SP plug-ins, returned in *pluginList*. Plug-in identification strings such as their host adapter are searched for in the specified *stringPool*.

You can also use Suite PEA's main plug-in list, available through the SPRuntime suite. See the discussion on Suite PEA lists in the chapter, "The Suite PEA Core."

FreePluginList( ) — Dispose of an plugin list

```
SPAPI SPErr FreePluginList( SPPluginListRef pluginList );
```

Disposes of the specified plugin list and any entries in it. The plugin list was created with the `AllocatedPluginList( )` function.

See the discussion on Suite PEA lists in the chapter, "The Suite PEA Core."

AddPlugin( ) — Add a new an plug-in to a list

```
SPAPI SPErr SPAddPlugin( SPPluginListRef pluginList,
            SPPlatformFileSpecification *fileSpec, PIPropertyList
            *PiPL, char *adapterName, void *adapterInfo, SPPluginRef
            *plugin );
```

Adds a plug-in to the indictated *pluginList*. Pass NULL to use Suite PEA's main plug-in list. The *fileSpec* is a reference file containing the plug-in code and resources. The *PiPL* is a pointer to a structure containing the plug-in's properties. The *adapterName* is an identifier for the adapter adding the plug-in. When a plug-in is to be called, this string is to identify their host and thus the correct way to call it. The *adapterInfo* is used by an adapter adding the plug-in to store relvant information about the it. The contents of the structure to which it points are determined by the adding adapter and might contain version information, options, or other data. A reference to the added *plugin* will be returned.

NewPluginListIterator( ) — Create an iterator to traverse a list

```
SPAPI SPErr NewPluginListIterator( SPPluginListRef pluginList,
            SPPluginListIteratorRef *iter );
```

Returns an iterator to access an plug-in list. The iterator is set to the first plug-in in the list.

See the discussion on Suite PEA lists in the chapter, "The Suite PEA Core."

NextPlugin( ) — Advance to the next entry in a list

```
SPAPI SPErr NextPlugin( SPPluginListIteratorRef iter,
            SPPluginRef *plugin );
```

Advances the list iterator to the next plug-in in the list. When the last plug-in has been reached, this function will return NULL.

See the discussion on Suite PEA lists in the chapter, "The Suite PEA Core."

DeletePluginListIterator( ) — Dispose of an plug-in list iterator

```
SPAPI SPErr DeletePluginListIterator( SPPluginListIteratorRef iter );
```

When a list iterator is no longer needed it should be disposed.

See the discussion on Suite PEA lists in the chapter, "The Suite PEA Core."

GetHostPluginEntry( ) — Get the entry point of a host supplied plug-in

```
SPAPI SPErr GetHostPluginEntry( SPPluginRef plugin, void **host );
```

If the *plugin* has added directly by the host application, this function returns the address of the entry point function in *host*. If the plug-in is a normal, standalone file, *host* will be nil.

While most plug-ins are files seperate from the application, Suite PEA allows its host application to add plug-ins directly to it. Host plug-ins have the same entry point definition and are sent the same messages as normal plug-ins. The main difference is that they aren't unloaded from memory. The host might use a host plug-in to add its suites or an adapter for an earlier version of its API.

GetPluginFileSpecification( ) — Get the file specification of a plug-in

```
SPAPI SPErr GetPluginFileSpecification( SPPluginRef plugin, SPPlat-
            formFileSpecification *fileSpec );
```

Files in the SPPlatformFileSpecification *fileSpec* with the location of the plug-in file.

A plug-in might want to know its location on the disk to access support files. See the SPFiles suite chapter for more information.

GetPluginPropertyList( ) — Get the property list of a plug-in

```
SPAPI SPErr GetPluginPropertyList( SPPluginRef plugin, SPPropertyLis-
            tRef *propertyList );
```

Returns a reference to the plug-in's property list in *propertyList*.

See the SPProperties suite and Suite PEA PiPLs chapters for more information on properties.

See also: FindPluginProperty( )

GetPluginGlobals( )

Return the globals reference stored for a plug-in

```
SPAPI SPErr GetPluginGlobals( SPPluginRef plugin, void **globals );
```

Suite PEA stores a 4 byte value for a plug-in in case it is unloaded. This function returns that value for the specified plug-in.

This is the same value passed in the globals field of the SPMessageData structure.

See also: SetPluginGlobals( )

SetPluginGlobals( )

Set the globals reference stored for a plug-in

```
SPAPI SPErr SetPluginGlobals( SPPluginRef plugin, void *globals );
```

Sets the value of a plug-in stored globals data to **globals*.

Suite PEA and any plug-in that calls another should store the returned globals value in the SPMessageData structure in case it has been changed.

Usually the value of the SPMessageData field *globals* is a pointer to a globals structure allocated once at startup. A globals structure could be reallocated and the new pointer returned. Also, if the plug-in only needs four bytes of data, the data could be that 4 byte value.

See also: GetPluginGlobals( )

GetPluginStarted( )

Return a plug-in's started state

```
SPAPI SPErr GetPluginStarted( SPPluginRef plugin, long *started );
```

Returns whether or not the plug-in has been started. True will be returned if the plug-in has been sent the interface/startup message.

See also: SetPluginStarted( )

SetPluginStarted( )

Set a plug-in's started state

```
SPAPI SPErr SetPluginStarted( SPPluginRef plugin, long started );
```

Sets whether the plug-in has been started. This value is set to true if the plug-in has been sent the interface/startup message.

A plug-in adapter would use this function; it is unlikely that other plug-ins would.

GetPluginBroken( ) — Return a plug-in's broken state

```
SPAPI SPErr GetPluginBroken( SPPluginRef plugin, long *broken );
```

Returns whether or not the plug-in is broken.

A broken setting of true indicates that a condition has occured that makes the plug-in unavailable, for instance, its file being removed from the plug-ins folder. A plug-in adapter would want to check this condition before it calls one of its plug-ins.

See also: SetPluginBroken( )

SetPluginBroken( ) — Set a plug-in's broken state

```
SPAPI SPErr SetPluginBroken( SPPluginRef plugin, long broken );
```

Sets whether or not the plug-in is broken.

A broken setting of true indicates that a condition has occured that makes the plug-in unavailable, for instance, its file being removed from the plug-ins folder. A plug-in adapter would set this condition if it encounters such a situation; other plug-in are unlikely to use this function.

See also: SetPluginBroken( )

GetPluginAdapter( ) — Return a plug-in's host adapter

```
SPAPI SPErr GetPluginAdapter(SPPluginRef plugin, SPAdapterRef *adapt-
            er);
```

Gets a reference to the adapter of a plug-in.

This reference could be used to get the adapter name, allowing it to be called correctly. See the description of GetAdapterName( ) for an example.

See also: GetAdapterName( )

GetPluginAdapterInfo( ) — Return a plug-in's host adapter info

```
SPAPI SPErr GetPluginAdapterInfo(SPPluginRef plugin, void **adapter-
            Info);
```

Gets the adapter information of a plug-in. A pointer to the *plugin*'s adapter's data structure is returned in *adapterInfo*.

This function is unlikely to be used by plug-ins other than adapters, since the contents of the structure pointed to by adapterInfo are only known to it. Other plug-ins are more likely to use an adapter provided suite of access functions. For instance, the Adobe Illustrator application and adapters provide these functions to access plug-in information:

```
    AIAPI AIErr (*GetPluginOptions) ( SPPluginRef plugin, long
*options );
```

```
    AIAPI AIErr (*SetPluginOptions) ( SPPluginRef plugin, long
options );
```

See also: SetPluginAdapterInfo( )

SetPluginAdapterInfo( )

Set a plug-in's host adapter info

```
SPAPI SPErr SetPluginAdapterInfo(SPPluginRef plugin, void *adapterIn-
            fo);
```

Sets the adapter information of a plug-in. The pointer *adapterInfo* to the adapter's data structure is stored with the indicated *plugin*.

This function is unlikely to be used by plug-ins other than adapters, since the contents of the structure pointed to by adapterInfo are only known to it.

See also: GetPluginAdapterInfo( )

FindPluginProperty( )

Find a property in a plug-in's property list

```
SPAPI SPErr FindPluginProperty( SPPluginRef plugin, PIType vendorID,
            PIType propertyKey, long propertyID, PIProperty **p );
```

This function searches the indicated *plugin*'s property list for the property specified by *vendorID*, *propertyKey*, and *propertyID*. If found, a pointer to the property is returned in *p*.

If Suite PEA doesn't find the property in the property list, it asks the plug-in for it, with a properties caller/acquire property message. The plug-in may ignore the request or it may manufacture the property and return it to Suite PEA. In either case, Suite PEA adds the new (possibly NULL) property to the plug-in's property list. FindPluginProperty( ) will find the stored property on subsequent calls without having to ask the plug-in (that is, it only asks the plug-in once for a particular property).

Interpreting the property data is up to the plug-in calling the function. More information on properties is found in the chapter "Suite PEA PiPLs".

13 The SPProperties Suite

About the SPProperties Suite

Unlike PICA's global lists, a properties list is associated with a plug-in. Each plug-in's list of properties is accessed with the SPProperties suite. If PICA, the host application, or a plug-in needs to examine the elements of a plug-in's PiPL for some reason or find a particular property, it would use the functions in this suite rather than repeatedly parsing the plug-ins property resource. Properties are described in detail in the PICA Properties chapter.

Accessing the Suite

The SPProperties suite is referred to as:

```
#define kSPPropertiesSuite "SP Properties Suite"
```

with the version constant:

```
#define kSPPropertiesSuiteVersion 3
```

It is acquired with the SPBasic suite as follows:

```
SPPropertiesSuite *sSPProperties;
error = sSPBasic->AcquireSuite( kSPPropertiesSuite,
                          kSPPropertiesSuiteVersion, &sSPProperties
);
if ( error ) goto error;
```

SPProperties

Each property of a plug-in is added to the plug-ins SPProperties list. The list can be iterated and references to the added properties accessed. Entries in a property list, SPProperties, include the PIProperty structure and some additional information: a four-byte refcon value and a flag denoting whether the property information is cacheable. The refcon can be used to store additional information about the property when it is added. The cacheable flag indicates properties that will not change, so that the host can keep a cache file of properties rather than read them from each plug-in file at startup.

SPProperties Suite Functions

AllocatePropertyList( ) — Create a new property list

```
SPAPI SPErr AllocatePropertyList( SPPropertyListRef *propertyList );
```

Allocates a new list of SP properties, returned in *propertyList*.

Unlike PICA's other lists, property lists are not global collections available through the SPRuntime suite. Instead they are associated with a plug-in and the returned value of this function would likely be assigned to a plug-in.

See the discussion on PICA lists in the chapter, "The Core."

FreePropertyList( ) — Dispose of an property list

```
SPAPI SPErr FreePropertyList( SPPropertyListRef propertyList );
```

Disposes of the specified property list and any entries in it. The property list was created with the AllocatedPropertyList( ) function.

See the discussion on PICA lists in the chapter, "The Core."

AddProperties( ) — Add a list of properties

```
SPAPI SPErr AddProperties( SPPropertyListRef propertyList,
            PIPropertyList *pList, long refCon, long cacheable );
```

Adds a list of properties, referenced by *plist,* to the indicated *propertyList*. The *refCon* is a value which you can assign. The *cacheable* argument indicates whether or not the property list is likely to change. Cacheable properties may be written to a cache file kept by the host rather than read from the plug-in file. Each of the properties in the list is added as an `SPProperyRef`.

The PIPropertyList would likely be read from a resource and then assigned with this function.

AddProperty( ) — Add a property to a list

```
SPAPI SPErr AddProperty( SPPropertyListRef propertyList, PIType ven-
            dorID, PIType propertyKey, long propertyID, PIProperty
            *p,
            long refCon, long cacheable, SPPropertyRef *property );
```

Adds a single property to the indicated *propertyList*. The property is described by the *vendorID*, *propertyKey*, *propertyID*, and property pointer *p*, which are as described in the PICA properties chapter. The *refCon* is a value which you can assign. The *cacheable* argument indicates whether or not the property list is likely to change. Cacheable properties may be written to a cache file kept by the host rather than read from the plug-in file. A reference to the added property is returned in *property*.

This function would be used, for instance, to assign a property returned by an acquire properties messages sent to a plug-in.

FindProperty( ) — Find a property

```
SPAPI SPErr FindProperty( SPPropertyListRef propertyList,
            PIType vendorID, PIType propertyKey, long propertyID,
            SPPropertyRef *property );
```

Get a reference to the property indicated by the *vendorID*, *propertyKey*, *propertyID*. The *propertyList* is searched for the matching property and if found, it is returned in *property*. Otherwise NULL is returned.

See the discussion on PICA lists in the chapter, "The Core."

NewPropertyListIterator( ) — Create an iterator to traverse a list

```
SPAPI SPErr NewPropertyListIterator( SPPropertyListRef propertyList,
            SPPropertyListIteratorRef *iter );
```

Returns an iterator to access a property list. The iterator is set to the first property in the list.

See the discussion on PICA lists in the chapter, "The Core."

NextProperty( ) — Advance to the next entry in a list

```
SPAPI SPErr NextProperty( SPPropertyListIteratorRef iter,
            SPPropertyRef *property );
```

Advances the list iterator to the next property in the list. When the last property has been reached, this function will return NULL in *property*.

See the discussion on PICA lists in the chapter, "The Core."

DeletePropertyListIterator( ) — Dispose of an property list iterator

```
SPAPI SPErr DeletePropertyListIterator( SPPropertyListIteratorRef
            iter );
```

When a list iterator is no longer needed it should be disposed.

See the discussion on PICA lists in the chapter, "The Core."

GetPropertyPIProperty( ) — Get the plug-in hosting an adapter

```
SPAPI SPErr GetPropertyPIProperty(SPPropertyRef property, PIProperty
            **p);
```

Returns a pointer, *p*, to the PIProperty structure for the indicated *property* reference.

This structure is described in the "PiPLs" chapter.

GetPropertyRefCon( ) — Get the property refcon

```
SPAPI SPErr SPGetPropertyRefCon( SPPropertyRef property, long *ref-
            Con );
```

Get the SPProperty refcon, which is assigned when the property is added. The meaning of this value is undefined, determined by who set it.

GetPropertyCacheable( ) — Get whether the property is cacheable

```
SPAPI SPErr GetPropertyCacheable( SPPropertyRef property,
            long *cacheable );
```

Get whether the property is cacheable.

Cacheable properties are static and will not change. Because of this, cacheable properties may be stored by the host in a cache file rather than read from the plug-in file.

GetPropertyAllocatedByPlugin( ) — Get whether or not the property was allocated by the plug-in

```
SPAPI SPErr GetPropertyAllocatedByPlugin( SPPropertyRef property,
            long *allocatedByPlugin );
```

Get whether or not the property was allocated by the plug-in. If the property was provided by a plug-in in response to a acquire properties message, this flag is true. If the property was read from a resource, this flag is false.

14 The SPRuntime Suite

About the SPRuntime Suite

The functions in this suite are used to request references to the main data strucures (it lists and stringpool) used by the PICA plug-in manager. The data structure suite functions take a reference to a list or pool, and to use the appropriate PICA list, you would use a reference returned by one this suite's functions. That said, as a short cut you can pass NULL for a data structure reference and PICA will use its list as a default.

See the chapter "The PICA Core" for more information on data structures and their use.

Accessing the Suite

The SPRuntime Suite is referred to as:

```
#define kSPRuntimeSuite "SP Runtime Suite"
```

with the version constant:

```
#define kSPRuntimeSuiteVersion 2
```

It is acquired with the SPBasic suite as follows:

```
SPRuntimeSuite *sSPRuntime;
error = sSPBasic->AcquireSuite( kSPRuntimeSuite,
                          kSPRuntimeSuiteVersion, &sSPRuntime);
if ( error ) goto error;
```

SPRuntime Suite Functions

GetRuntimeStringPool( ) — Get PICA's string pool

```
SPAPI SPErr GetRuntimeStringPool( SPStringPoolRef *stringPool );
```

Returns a reference to the string pool used by PICA to store suite names, adapter names, and other strings.

GetRuntimeSuiteList( ) — Get PICA's suite list

```
SPAPI SPErr GetRuntimeSuiteList( SPSuiteListRef *suiteList );
```

Returns a reference to the suite list used by PICA to store its suites, including those added by the application and plug-ins.

GetRuntimeFileList( ) — Get PICA's file list

```
SPAPI SPErr GetRuntimeFileList( SPFileListRef *fileList );
```

Returns a reference to the file list used by PICA. This list contains all the files in the application specified plug-in folder and its sub-directories.

GetRuntimePluginList( ) — Get PICA's plug-in list

```
SPAPI SPErr GetRuntimePluginList( SPPluginListRef *pluginList );
```

Returns a reference to the plug-in list used by PICA. This list contains all the PICA plug-ins and any added by an adapter.

GetRuntimeAdapterList( ) — Get PICA's adapter list

```
SPAPI SPErr GetRuntimeAdapterList( SPAdapterListRef *adapterList );
```

Returns the list of plug-in adapters used by PICA. This list will have PICA's built in adapter and any added by plug-ins.

GetRuntimeHostProcs( ) — Get PICA's host supplied functions

```
SPAPI SPErr GetRuntimeHostProcs( SPHostProcs **hostProcs );
```

When PICA is initialized by the host application, it is supplied a block of function pointers for host provided functionality. This allows it to use the application's resources, some of which are exposed to plug-ins. These include memory routines (SPBlocks), notification routines, exception handling, and string pool routines (SPStrings). The function pointers can be accessed with `GetRuntimeHostProcs( )`.

A plug-in isn't likely to need the host functions, and instead will find it easier to use the PICA suites. An exception to this is if the plug-in provides an API adapter. In this case it will want to use two of the host functions, `startupNotify( )` and `shutdownNotify( )`, which are defined as:

```
typedef void (*SPStartupNotifyProc)( SPPluginRef plugin, void *host-
Data );
typedef void (*SPShutdownNotifyProc)( SPPluginRef plugin, void *host-
Data );
```

The startup notification function is called by the adapter to inform it that a plug-in is being started up. The plug-in and any data specified by the host is passed by the function. The host application will use this to track the startup process. One possible use of the plug-in startup notification is to display to the user a list of plug-ins being loaded.

The shutdown notification sends the application the same information, but when the plug-in is being shutdown. Its purpose is for the host application to track the shutdown process, possibly providing additional feedback for the user.

15 The SPStrings Suite

About the SPStrings Suite

The functions in this suite are used to access string pools and their strings. String pools can be created and deleted. Strings can be added to the pool.

Accessing the Suite

The SPStrings Suite is referred to as:

```
#define kSPStringsSuite "SP Strings Suite"
```

with the version constant:

```
#define kSPStringsSuiteVersion 2
```

It is acquired with the Basic Suite as follows:

```
SPStringsSuite *sSPStrings;
error = sSPBasic->AcquireSuite( kSPStringsSuite,
                          kSPStringsSuiteVersion, &sSPStrings);
if ( error ) goto error;
```

String Pools

A string pool is a simple string atomizer, providing an efficient central storage space for C style strings. When a string is placed in the pool, PICA first checks to see if it already exists. If so, a pointer to the existing pooled string will be returned. If not, the string pool manager will copy the string into the string pool and return a pointer to the copy.

The idea behind a string pool is that string equality tests can be reduced to pointer equality tests if each string exists in one and only one place. To compare two string pooled strings, you simply compare the pointers. If they are the same, the strings are the same.

For example, if you store all suite names in a string pool, then looking for a suite by name involves atomizing the name for which you're looking and then comparing the atomized string pointer to the atomized string pointers in the suite list. If the search name is in the list, the process of atomizing it will give you the same string pointer.

SPStrings Suite Functions

AllocateStringPool( ) — Allocate a new string pool

```
SPAPI SPErr AllocateStringPool( SPStringPoolRef *stringPool );
```

This function creates a string pool reference and allocates an intial block of memory for its strings.

FreeStringPool( ) — Release memory occupied by a string pool

```
SPAPI SPErr FreeStringPool( SPStringPoolRef stringPool );
```

The function frees any memory used by a string pool and makes the pool reference invalid.

MakeWString( ) — Add a string to a string pool

```
SPAPI SPErr MakeWString( SPStringPoolRef stringPool, char *string,
            char **wString );
```

This function adds *string* to the indicated *stringPool* and returns an atomized string pointer, *wString*. If the string doesn't exist in the pool then SPMakeWString( ) adds it and returns the new address. If the string has already been added to the pool, a reference to the existing pooled string will be returned.

```
char *gMyPooledName;

sSPStrings->MakeWString( nil, "Buckwheat", &gMyPooledName);
```

Note: In case you're wondering, the 'W' means 'wet'. If you put a string in a pool, it is a wet string.

16 The SPSuites Suite

About the SPSuites Suite

The SPSuites suite is used to access PICA's suite list and obtain information about entries in it. This list is created at startup and contains references to every suite added by PICA, the host application, and plug-ins.

Suites are a pointer to a data structure, usually containing function pointers. The functions generally haves some common purpose, such as accessing a data type, and are used by plug-ins to interact with PICA, the host application, and each other. In order to use a function in a suite, the suite must first be acquired. More general information on suites is found in the chapters "PICA Intro" and "The SPBasic Suite."

Functions to add new suites and acquire existing ones are found in the SPSuites API and discussed here. Other functions for getting information on a particular suite are also available.

Accessing the Suite

The SPSuites suite is referred to as:

```
#define kSPSuitesSuite "SP Suites Suite"
```

with the version constant:

```
#define kSPSuitesSuiteVersion 2
```

It is acquired with the basic suite as follows:

```
SPSuitesSuite *sSPSuites;
error = sSPBasic->AcquireSuite( kSPSuitesSuite,
                            kSPSuitesSuiteVersion, &sSPSuites);
if ( error ) goto error;
```

Plug-in Suites

The PICA API is made up of a number of function suites to manage a plug-in API. These are supplemented by the host application's suites, which a plug-in is more likely to use. A plug-in can extend the API further by adding its own suites. There is no difference between a suite supplied by the application and one supplied by a plug-in.

Inherent in the design of early Adobe monolithic APIs is the assumption that plug-ins are simple extensions of the application. Much as a central power grid provides electricity to the public, the host application was the sole provider of plug-in functionality. The single provider, one-way approach is unsuitable for a growing and diversifying interface.

Sweet Pea is designed so that plug-ins can provide functionality as well, putting electricity back into the system. Modern power systems allow for independent production of electricity, supplemented by or added to the

central power source. A plug-in suite can add entirely new functionality to the API for private or public use. It can correct errant behavior of an existing suite or extend its functionality. Systems can be designed where a plug-in suite replaces functionality of the host application with an alternate system.

To the acquirer, acquiring a suite provided by a plug-in is no different than acquiring one provided by the application or PICA. Host suites are never unloaded, though, while suites provided by plug-ins will be unloaded with the plug-in. Acquiring a suite may cause its proividing plug-in to be loaded into memory to make the suite available. A reference count is kept for each suite. The count is incremented when it is acquired and decremented when it is released. If a suite with a positive acquire count is provided by a plug-in, its plug-in will not be unloaded. When a plug-in suite's acquire count is 0, its plug-in can be unloaded.

Suite Versions

Public information about a suite provides a name and an api version number that define it. These are used with the SPBasic suite to acquire and release a reference to the suite. There is one additional piece of information which is used internally be PICA to define a suite: a subversion number.

The api (or major) version number indicates the functionality of a suite, indicating its general purpose and behavior, constants, data types and functions; in other words everything in the suite's header file. If a major change to the suite's definition or behavior is made, the api version number would changed. A simple rule of thumb is if the header file changes, then the API version number changes. Sweet Pea tests API version numbers only for equality, it assumes no continuum. That is, Sweet Pea does not think that version 10 is newer than version 9. It also does not interpret ranges of bits within the number. Suites are free to use any numbering scheme they want.

The internal, or implementation, version number is the way Sweet Pea chooses between multiple occurrences of the same suite with the same API version. It is intended for bug fixes or improvements in the implementation. For instance, if a suite fixed a bug in a previous version but all other aspects of it were the same, then the api version number would remain the same while the subversion number would be incremented. Sweet Pea interprets greater values as newer (better) versions. Sweet Pea provides a special constant that means the latest version. You can use this when looking for a suite.

```
#define kSPLatestInternalVersion  0
```

To succeed, a search for a suite must match the name and API version number exactly. If you ask for the latest internal version number (the common case), then it finds that. If you specify a real internal version number, then that too must match exactly.

Suite Interface Files

A suite's identifying name and api version constants should be available in a public header file. Because the names and versions must match exactly, a suite publisher must explicitly declare every version of a suite it supports. The internal version needs to be documented, but will likely not be public. Other definitions, such as error strings particular to a suite's functions, are also found in the include file. Finally, if the suite has related plug-in messages they will also be defined. At the end are the suite functions. The function pointers should be fully prototyped.

Supplying Multiple Suite Versions

For backward compatibility purposes it is desirable to provide multiple versions of a suite. The convention for doing this is to provide versioned constants for all api versions and then make one of them the default:

```
#define kAIPathStyleSuite             "AI Path Style Suite"
#define kAIPathStyleSuiteVersion2     2
#define kAIPathStyleSuiteVersion3     3
#define kAIPathStyleSuiteVersion      kAIPathStyleSuiteVersion3
```

There are many ways which a suite might be extended. One is to add functions to the end of the existing suite. This allows the provider to use common code for both versions. For more drastic changes it might be more appropriate to clone the suite's code base or start over from scratch. The old version of the suite is supplied by the original code ensuring its integrity. The new version is implemented in the new code, providing a fresh start.

There is no provision for specifying a range of versions. However, a suite publisher can use the same pointer when declaring multiple versions of its suite. For example, if the only change made to version 1 for version 2 is the addition of functions at the end of the structure, then both suites can point to it. Version 1 clients are unaware of the extra functions. Yet the capability exists for versions of a suite to differ radically. The suite's programmer may take the opportunity with a new version to rearrange the suite's functions or add parameters to existing functions.

Finally, Sweet Pea makes no assumptions about which plug-ins implement which versions of a suite. A different plug-in may implement each version. Several plug-ins may implement different ranges of version numbers, and the ranges need not be contiguous. For example, a plug-in may replace another plug-in's version with a newer implementation (internal version).

To reemphasize, Sweet Pea assumes no interpretation of a suite's contents or version number and assumes nothing about who implements it.

Adding and Allocating Suites

When a plug-in wants to make a function suite available it will use the SPSuites API made available by PICA. As with any suite, the plug-in must first acquire the SPSuites API using the SPBasic suite. To make a suite of functions available the AddSuite( ) function is used.

```
SPSuiteRef AddSuite( SPSuiteListRef suiteList, SPPluginRef host,
            char *name, long apiVersion, long internalVersion,
            void *suiteProcs, SPErr *error );
```

To add a suite to PICA's main suite list, pass NULL for the *suiteList* argument. The *name*, *apiVersion*, and *internalVersion* are the suite identifiers as already described. The *suiteProcs* argument is a pointer to the suite structure filled with function pointers.

The structure with the fully prototyped functions is a part of the public header, for instance:

```
#define kMyAppMenuSuite       "My App Menu Suite"
#define kMyAppMenuSuiteVersion1

typedef struct SPMyAppMenuSuite {
    SPAPI void (*AddMyMenu)( SPPluginRef self,
            MyAppAddMenuData theMenuData, MyAppMenu *theMenu );
} MyAppMenuSuite;
```

The plug-in allocates memory for this structure and then assigns its members valid function pointer. The memory for the structure should be an allocated block in the host's heap rather than a global since the plug-in may be unloaded. As an example:

```
#define kMyAppMenuSuiteInternalVersion1

MyAppMenuSuite *myAppMenuSuite;
SPSuiteRef myAppMenuSuiteRef;

SPErr AddMyAppMenuSuite( void ) {
    SPErr *error;

    sSPBlocks->AllocatedBlock( sizeof(MyAppMenuSuite), nil,
                                                &myAppMenuSuite);
    if (error) return error;
    myAppMenuSuite->AddMyMenu = AddMyMenu;

    error = sSPSuites->AddSuite( nil, nil, kMyAppMenuSuite,
        kMyAppMenuSuiteVersion, kMyAppMenuSuiteInternalVersion,
        &myAppMenuSuite, &myAppMenuSuiteRef );
}

AddMyMenu( SPPluginRef self, MyAppAddMenuData theMenuData,
            MyAppMenu *theMenu ) {
    // application code to add menu goes here
}
```

Loading and Unloading Suites

Because a plug-in can be loaded and unloaded, it must reinitialize the function pointers in the structure when it receives the access caller with unload and reload selectors.

Unloading the suite means stuffing the suite's procedure pointers with the address of the `Undefined( )` function in the SPBasic suite. This is a protective measure against other plug-ins that may mistakenly use the suite after they have released it. Reloading the suite means assigning the suite's procedure pointers with the new addresses of their functions.

```
SPErr UnloadSuite( MySuite *mySuite, SPAccessMessage *message ) {

    mySuite->functionA = (void *) message->d.basic->Undefined;
    mySuite->functionB = (void *) message->d.basic->Undefined;
}

SPErr ReloadSuite( MySuite *mySuite, SPAccessMessage *message ) {

    mySuite->functionA = functionA;
    mySuite->functionB = functionB;

}
```

Plug-in Suites, Export Properties, and Loading Order

A plug-in requests most suites not knowing which component of the application will provide them; the exception is the PICA core suites. If a plug-in requests a suite provided by an unloaded plug-in, or even one that has yet to be started, a potential conflict arises. It can do this safely however, relying on PICA to ensure that that they are available when needed.

PICA uses the export properties of plug-in to determine the location of all potential suite resources. When a suite is requested which is not yet in the suite list, PICA will scan the export properties of all plug-in searching for the name and version of the requested. All providers of the suites are sent a startup message. PICA then checks the list of plug-ins for the latest available version of the suite and returns this to the requesting plug-in.

When a requested suite is in the suite list, but provided by a plug-in that is currently unloaded from memory, PICA will acquire teh providing plug-in and send it a reload message. The providing plug-in reinitializes its suites' function pointers at the reload message. When it has returned from the reload call, the original requester will be sent the now valid suite.

An early version of the PICA technology found in Adobe Illustrator 6.0 handled the startup process in a different fashion. Before starting any plug-ins, it would acquire all export properties and a property of type import ('impt'). It would use these to build a dependency graph for the plug-ins and use this graph to load the plug-ins. Implementation-wise, either method is acceptable. PICA does not currently expect or recognize import properties from plug-ins.

SPSuites Suite Functions

AllocateSuiteList( ) — Create a new suite list

```
SPAPI SPErr AllocateSuiteList( struct SPStringPool *stringPool,
            SPSuiteListRef *suiteList );
```

Allocates a new list of PICA suites, returned in *suiteList*. Suite names added to the new suite list will be stored in the specified *stringPool*.

You can also use PICA's main suite list, available through the SPRuntime suite. See the discussion on PICA lists in the chapter, "The Core."

FreeSuiteList( ) — Dispose of an suite list

```
SPAPI SPErr FreeSuiteList( SPSuiteListRef suiteList );
```

Disposes of the specified suite list and any entries in it. The suite list was created with the `AllocatedSuiteList( )` function.

See the discussion on PICA lists in the chapter, "The Core."

AddSuite( ) — Add a new an suite to a list

```
SPAPI SPErr AddSuite( SPSuiteListRef suiteList, SPPluginRef host,
            char *name, long apiVersion, long internalVersion,
            void *suiteProcs, SPSuiteRef *suite );
```

This function adds a suite to the indicated *suiteList*. The plug-in providing the suite is specified in *host*. The *name, apiVersion*, and *internalVersion* identify the suite. A pointer to the structure with the suite pointers is passed for *suiteProcs*. A reference to the stored suite is returned in *suite*.

The suite list reference can be NULL, which indicates that the suite is to be added to PICA's main list. If you are keeping a private list of suites for some reason, this list reference could be passed instead. The identifers for the suite are discussed in the chapter introduction. The name and api version identifying constants should be placed in a public header file.

AcquireSuite( ) — Acquire a suite of functions

```
SPAPI SPErr AcquireSuite( SPSuiteListRef suiteList, char *name,
            long apiVersion, long internalVersion, void **suiteProcs
            );
```

A pointer to the requested suite is returned in *suiteProcs.* This differs from the SPBasic suite function of the same name in that the *suiteList* and *internalVersion* of the suite can be specified. Acquiring a suite increments its access count, preventing plug-in's that provide suites from being unloaded.

Passing 0 for the internal version argument will acquire the latest internal version available. Passing nil for the suite list argument will use PICA's internal list.

ReleaseSuite( ) — Find an suite by name

```
SPAPI SPErr ReleaseSuite( SPSuiteListRef suiteList, char *name,
            long apiVersion, long internalVersion );
```

Releasing a suite decrements its access count and makes its function pointers invalid. If the access counts of all suites provided by a single plug-in are 0, the plug-in may be unloaded.

FindSuite( ) — Find an suite by name

```
SPAPI SPErr SPFindSuite( SPSuiteListRef suiteList, char *name,
            long apiVersion, long internalVersion, SPSuiteRef *suite
            );
```

Get a reference to the suite identified by *name* and version information. The suiteList is search fro matching values and the pointer to the suite is returned in *suite*.

See the discussion on PICA lists in the chapter, "The Core."

NewSuiteListIterator( ) — Create an iterator to traverse a list

```
SPAPI SPErr NewSuiteListIterator( SPSuiteListRef suiteList,
            SPSuiteListIteratorRef *iter );
```

Returns an iterator to access an suite list. The iterator is set to the first suite in the list.

See the discussion on PICA lists in the chapter, "The Core."

NextSuite( ) — Advance to the next entry in a list

```
SPAPI SPErr NextSuite( SPSuiteListIteratorRef iter, SPSuiteRef
            *suite );
```

Advances the list iterator to the next suite in the list. When the last suite has been reached, this function will return NULL.

See the discussion on PICA lists in the chapter, "The Core."

DeleteSuiteListIterator( ) — Dispose of an suite list iterator

```
SPAPI SPErr DeleteSuiteListIterator( SPSuiteListIteratorRef iter );
```

When a list iterator is no longer needed it should be disposed.

See the discussion on PICA lists in the chapter, "The Core."

GetSuiteHost( )

Get the plug-in hosting a suite

```
SPAPI SPErr GetSuiteHost( SPSuiteRef suite,
            struct SPPlugin **plugin );
```

Returns a reference to the plug-in that added the indicated suite in *plugin*. You might want this reference to send the suite's plug-in a message.

GetSuiteName( )

Get the identifier string of a suite

```
SPAPI SPErr GetSuiteName( SPSuiteRef suite, char **name );
```

Get a pointer to the name of the indicated suite, which is one of three values used to identify it. The string should not be modified.

GetSuiteAPIVersion( )

Get the version of a suite

```
SPAPI SPErr GetSuiteAPIVersion( SPSuiteRef suite, long *version );
```

Get the api, or primary, version of the indicated suite, which is one of three values used to identify it.

GetSuiteInternalVersion( )

Get the internal version of a suite

```
SPAPI SPErr GetSuiteInternalVersion( SPSuiteRef suite, long *version
            );
```

Get the internal version, or sub-version, of the indicated suite, which is one of three values used to identify it.

GetSuiteProcs( )

Get the a pointer to the suite functions

```
SPAPI SPErr GetSuiteProcs( SPSuiteRef suite, void **suiteProcs );
```

Get a pointer to the block of memory containing the function of a suite. This is normally done by acquiring the suite.

Getting the suite functions in this manner does not increase the access count of the suite, nor will it cause the loading of the plug-in providing the suite. Because of this the function pointers in the memory block may be invalid. Before using them, check the suite access count with `GetSuiteAcquireCount( )`.

GetSuiteAcquireCount( )

Get the mumber of times a suite has been acquired

```
SPAPI SPErr GetSuiteAcquireCount( SPSuiteRef suite, long *count );
```

Get the number of times a suite has been acquired. While this count is positive, the suite is valid. If it is zero, its function pointers may be invalid.