API Reference

Netscape Gecko Plug-ins

Version 2.0

August 2002

Table of Contents

Preface

About This Guide

The *Netscape Gecko™ Plug-in API Reference* describes the application programming interfaces (APIs) for Netscape Gecko plug-ins and provides information about how to use these interfaces to create plug-ins for Netscape Gecko-based browsers such as Netscape 6.x, Netscape 7.0, and Mozilla.

The general introduction in "**Plug-in Basics**" and a chapter entitled "**Plug-in Development Overview**" begin the guide. A series of chapters on specific programming topics such as "**Initialization and Destruction**", "**Streams**", and "**URLs**" provide more detail about the technical aspects and techniques for creating and managing plug-ins.

The API reference in the second half of the guide is divided up into two main halves, "**Plug-in Side Plug-in API**" and "**Browser Side Plug-in API**", as well as additional reference material about "**Structures**" and "**Constants**".

The guide is structured so the developers new to Netscape Gecko plug-ins can familiarize themselves with the APIs and particular aspects of the development process (e.g., "**Drawing and Event Handling**"), but so that plug-in developers can quickly access the API reference material they need.

Who Should Read This Guide

The *Netscape Gecko Plug-in API Reference* is meant for plug-in developers. Though it provides a basic overview of plug-ins and how they work in the browser, the guide presumes that you understand how plug-ins work, how they handle and display media, and what the basic architecture of a browser is that supports the use of plug-in software.

The guide also presumes that you know how to use application programming interfaces, have experience developing browser software such as plug-ins, components, or add-ons, and are familiar with C/C++, the language(s) in which the libraries were actually created and in which all of the examples are given.

The Plug-in Software Development Kit

A Plug-in software development kit (SDK) is available for Netscape Gecko plug-in developers. This SDK is located in the Mozilla source under `mozilla/modules/plugin/tools/sdk`. You can use it within the Mozilla source tree, or you can build it there and then use it outside of tree if you'd rather. In either case, the Mozilla source tree is required to get started developing plug-ins. You can also view the plug-in SDK samples and source code using the web-based source viewer:

http://lxr.mozilla.org/seamonkey/source/modules/plugin/tools/sdk/

The SDK is based on the API developed originally for Netscape browsers starting with Netscape 2.x. Some additions were made when Netscape 3.x and Netscape 4.x were released. The present SDK reflects major changes related to Mozilla code base: LiveConnect for plugin scriptability is no longer supported, existing plugins should be modified slightly to become scriptable again; the browser services are now accessible from the plug-in through the access to the service manager.

The SDK is intended to help in creating full-blown plugins to work with the Mozilla code base without actually having the whole Mozilla source tree present and built.

The Common folder contains stub implementations of the NPAPI methods. There is no need to modify files in this folder, just include them into your project. This is not necessary though, some samples or plugin projects may use their own implementations, the files in this folder are just an illustration of one possible way to do that.

The Samples section at this point contains the following plug-in samples:

Basic plug-in

Shows the bare bones of the plugin DLL. It does not do anything more than a 'Hello, World' for plug-ins. The basic plug-in demonstrates how the plugin DLL is invoked and how NPAPI methods are called. It can be used as a starting template for writing your own plug-in.

Simple plug-in

This plugin example illustrates specific for Mozilla code base features. It is scriptable via JavaScript and uses services provided by the browser.

XPCOM interfaces are implemented in the simple plug-in so the Mozilla browser is aware of its capabilities. The plugin does not draw in the native window but rather uses JavaScript box to display the result of its work. Therefore, there are no separate projects for different platforms in this sample.

Scriptable plug-in

Yet another example of plug-in scriptability. The scriptable plug-in implements two native methods callable from the JavaScript, and it draws in a native window, so it uses different projects for the different major platforms.

Windowless plug-in

The windowless plug-in is an example of a plugin which does not use native window messaging mechanism and relies exclusively on `NPP_HandleEvent` to receive GUI messages for painting and other tasks. This plug-in simply draws a gray rectangle in the occupied area.

Scriptable plug-in samples require that you generate cross-platform type library (XPT) files and place them in the Mozilla Plugins directory along with the plug-in DLLs. (For backwards compatibility with pre-Mozilla 1.0 and Netscape 6.x browsers, you must put the type library file in the Components directory instead. For more information on type libraries and plug-in path information, see **Plug-in Detection** in the “Plug-in Basics” chapter.)

To verify that Mozilla is aware of new .xpt files, you can look in the generated file, *xpti.dat*, where type libraries are listed. If you need to, you can call `netscape.plugins.refresh()` to to find new XPT files and plug-in software.

Plug-in developers might find it useful for debugging purporses to turn off the exception catching mechanism currently implemented in Mozilla on Windows. To turn off Windows exception handling, add the following line into your prefs.js file:

```
user_pref("plugin.dont_try_safe_calls", true);
```

Plug-in Basics

How Plug-ins Are Used

Plug-ins offer a rich variety of features that can increase the flexibility of Netscape Gecko-based browsers. Plug-ins like these are now available:

- multimedia viewers such as Macromedia Shockwave and Adobe Acrobat
- utilities that provide object embedding and compression/decompression services
- applications that range from personal information managers to games

The range of possibilities for using plug-in technology seems boundless, as shown by the growing numbers of independent software vendors who are creating new and innovative plug-ins.

With the Plug-in API, you can create dynamically loaded plug-ins that can:

- register one or more MIME types
- draw into a part of a browser window
- receive keyboard and mouse events
- obtain data from the network using URLs
- post data to URLs
- add hyperlinks or hotspots that link to new URLs
- draw into sections on an HTML page
- communicate with Javascript/DOM from native code

You can see which plug-ins are installed on your system and have been properly associated with the browser by consulting the Installed Plug-ins page. Go to the Help menu, and click Help and then About Plug-ins. The Installed Plug-ins page lists each installed plug-in along with its MIME type or types, description, extensions, and the current state (enabled or disabled) of the plug-in for each MIME type assigned to it. Notice in view-source that this information is simply gathered from the DOM.

Because plug-ins are platform-specific, you must port them to every operating system and processor platform upon which you want to deploy your plug-in.

Plug-ins and Helper Applications

Before plug-ins, there were helper applications. A helper application is a separate, free-standing application that can be started from the browser. Like a plug-in, the browser starts a helper application when the browser encounters a MIME type that is mapped to it. Unlike a plug-in, a helper application runs separately from the browser in its own application space and does not interact with the browser or the web.

When the browser encounters a MIME type, it always searches for a registered plug-in first. If there are no matches for the MIME type, it looks for a helper application.

Plug-ins and helper applications fill different application needs. For more information about helper applications, refer to the Netscape online help.

How Plug-ins Work

The life cycle of a plug-in, unlike that of an application, is completely controlled by the web page that calls it. This section gives you an overview of the way that plug-ins operate in the browser.

When Netscape Gecko starts, it checks for plug-in modules in the plugins directory or Plug-ins folder (Mac OS) located in the same folder or directory as the browser application. For more information, see "**How Netscape Gecko Finds Plug-ins**"

When the user opens a page that contains embedded data of a media type that invokes a plug-in, the browser responds with the following sequence of actions:

- check for a plug-in with a matching MIME type
- load the plug-in code into memory
- initialize the plug-in
- create a new instance of the plug-in

Netscape Gecko can load multiple instances of the same plug-in on a single page, or in several open windows at the same time. If you are browsing a page that has several embedded real audio clips, for example, the browser will create as many instances of the RealPlayer plug-in as are needed (though of course playing several real audio files at the same time would seldom be a good idea). When the user leaves the page or closes the window, the plug-in instance is deleted. When the last instance of a plug-in is deleted, the plug-in code is unloaded from memory. A plug-in consumes no resources other than disk space when it is not loaded. The next section, **Understanding the Runtime Model**, describes these stages in more detail.

Understanding the Runtime Model

Plug-ins are dynamic code modules that are associated with one or more MIME types. When the browser starts, it enumerates the available plug-ins (this step varies according to platform), reads resources from each plug-in file to determine the MIME types for that plug-in, and registers each plug-in library for its MIME types.

The following stages outline the life of a plug-in from loading to deletion:

- When Netscape Gecko encounters data of a MIME type registered for a plug-in (either embedded in an HTML page or in a separate file), it dynamically loads the plug-in code into memory, if it hasn't been loaded already, and it creates a new instance of the plug-in.

  Netscape Gecko calls the plug-in API function **NP_Initialize**[1] when the plug-in code is first loaded. By convention, all of the plug-in specific functions have the prefix “NPP”, and all of the browser-specific functions have the prefix “NPN”
- The browser calls the plug-in API function **NPP_New** when the instance is created. Multiple instances of the same plug-in can exist (a) if there are multiple embedded objects on a single page, or (b) if several browser windows are open and each displays the same data type.
- A plug-in instance is deleted when a user leaves the instance's page or closes its window; Netscape Gecko calls the function **NPP_Destroy** to inform the plug-in that the instance is being deleted.
- When the last instance of a plug-in is deleted, the plug-in code is unloaded from memory. Netscape Gecko calls the function **NP_Shutdown**. Plug-ins consume no resources (other than disk space) when not loaded.

NOTE: *Plug-in API calls and callbacks use the main Navigator thread. In general, if you want a plug-in to generate additional threads to handle processing at any stage in its lifespan, you should be careful to isolate these from Plug-in API calls.*

See “**Initialization and Destruction**” for more information about using these methods.

1. Note that `NP_Initialize` and `NP_Shutdown` are not technically a part of the function table that the plug-in hands to the browser. The browser calls them when the plug-in software is loaded and unloaded. These functions are exported from the plug-in DLL and accessed with a system table lookup, which means that they are not related to any particular plug-in instance. Again, see “**Initialization and Destruction**” for more information about initializing and destroying plug-ins.

Plug-in Detection

Netscape Gecko looks for plug-ins in various places and in a particular order. The next section, "**How Netscape Gecko Finds Plug-ins**," describes these rules, and the following section, "**Checking Plug-ins by MIME Type**," describes how you can use JavaScript to locate plug-ins yourself and establish which ones are to be registered for which MIME types.

How Netscape Gecko Finds Plug-ins

When a Netscape Gecko-based browser starts up on Windows or Unix systems, it checks for plug-in modules in the path pointed to by `MOZ_PLUGIN_PATH`. After that, it checks in the plug-ins directory for the platform:

- MS Windows: `plugins` subdirectory, in the same directory as the browser application.
- Mac OS: `Plug-ins` folder. A Mac OS plug-in can reside in a different directory if you install a Macintosh alias that links to the plug-in in the Plug-ins folder.
- Unix: `usr/local/lib/netscape/plugins` or `$HOME/.mozilla/plugins`. If you want to use a different directory, set the `MOZ_PLUGIN_PATH` environment variable to its filepath, for example, `$HOME/yourplugins:/usr/local/lib/netscape/plugins`. Netscape Gecko searches any directory that this variable specifies. The local user location, if it exists, overrides the network location.
- Finally, on the Mac, the browser scans the `~/Library/Internet Plugins` then `/Library/Internet Plugins`. Within these directories, the plug-ins are ordered by date.

On all platforms, the `plug-ins` subdirectory or folder must be in the same directory as the browser application. Users can install plug-ins in this directory manually, by using a binary installer program, or by using the XPInstall API to write an installation script, which the browser then uses to perform the installation. For more information about these options, see **Installing Plug-ins**.

To find out which plug-ins are currently installed, choose About Plug-ins from the Help menu (MS Windows and Unix) or "?" (Help) menu (Mac OS). Netscape Gecko displays a page listing all installed plug-ins and the MIME types they handle, as well as optional descriptive information supplied by the plug-in.

On Windows, installed plug-ins are automatically configured to handle the MIME types that they support. If multiple plug-ins handle the same MIME type, the *first plug-in registered* handles the MIME type. For information about the way MIME types are assigned, see **Registering Plug-ins**.

Checking Plug-ins by MIME Type

The `enabledPlugin` property in JavaScript can be used to determine which plug-in is configured for a specific MIME type. Though plug-ins may support multiple MIME types and each MIME type may be supported by multiple plug-ins, only one plug-in can be configured for a MIME type. The `enabledPlugin` property is a reference to a Plugin object that represents the plug-in that is configured for the specified MIME type.

You might need to know which plug-in is configured for a MIME type, for example, to dynamically create an `OBJECT` tag on the page if the user has a plug-in configured for the MIME type.

The following example uses the DOM to determine whether the Shockwave plug-in is installed. If it is, a movie is displayed.

```
// Can we display Shockwave movies?
mimetype = navigator.mimeTypes["application/x-director"]
if (mimetype) {
   // Yes, so can we display with a plug-in?
   plugin = mimetype.enabledPlugin
   if (plugin)
      // Yes, so show the data in-line
      document.writeln("Here\'s a movie:
         <OBJECT DATA=mymovie.dir HEIGHT=100 WIDTH=100>")
      else
      // No, so provide a link to the data
      document.writeln("<A HREF='mymovie.dir'>
         Click here</A> to see a movie.")
   } else {
   // No, so tell them so
   document.writeln("Sorry, can't show you this movie.")
}
```

Overview of Plug-in Structure

This section is an overview of basic information you will need as you develop plug-ins.

- **Understanding the Plug-in API**
- **Plug-ins and Platform Independence**

Understanding the Plug-in API

A plug-in is a native code library whose source conforms to standard C syntax. The Plug-in Application Programming Interface (API) is made up of two groups of functions and a set of shared data structures.

- Plug-in methods are functions that you implement in the plug-in; Netscape Gecko calls these functions. The names of all the plug-in functions in the API begin with `NPP_`, for example, `NPP_New`. There are also a couple of functions (i.e., `NP_Initialize` and `NP_Shutdown`), that are direct library entry points and not related to any particular plug-in instance.
- Browser methods are functions implemented by Netscape Gecko; the plug-in calls these functions. The names of all the browser functions in the API begin with `NPN_`, for example, `NPN_Write`.
- Data structures are plug-in-specific types defined for use in the Plug-in API. The names of structures begin with `NP`, for example, `NPWindow`.

All plug-in names in the API start with `NP`. In general, the operation of all API functions is the same on all platforms. Where this varies, the reference entry for the function in the reference section describes the difference.

Plug-ins and Platform Independence

A plug-in is a dynamic code module that is native to the specific platform on which the browser is running. It is a code library, rather than an application or an applet, and runs only from the browser. Although plug-ins are platform-specific, the Plug-in API is designed to provide the maximum degree of flexibility and to be functionally consistent across all platforms. This guide notes platform-specific differences in coding for the MS Windows, Mac OS, and Unix platforms.

You can use the Plug-in API to write plug-ins that are media type driven and provide high performance by taking advantage of native code. Plug-ins give you an opportunity to seamlessly integrate platform-dependent code and enhance the Netscape Gecko core functionality by providing support for new data types.

The plug-in file type depends on the platform:

- MS Windows: .DLL (Dynamic Link Library) files
- Unix: .SO or .DSO (Shared Objects) files
- Mac OS: PowerPC Shared Library files.

Windowed and Windowless Plug-ins

You can write plug-ins that are drawn in their own native windows or frames on a web page. Alternatively, you can write plug-ins that do not require a window to draw into. Using windowless plug-ins extends the possibilities for web page design and functionality. Note, however, that plug-ins are windowed by default, as windowed plug-ins are in general easier to develop and more stable to use.

- A windowed plug-in is drawn into its own native window on a web page. Windowed plug-ins are opaque and always come to the top HTML section of a web page.
- A windowless plug-in need not be drawn in a native window; it is drawn in its own drawing target. Windowless plug-ins can be opaque or transparent, and can be invoked in HTML sections.

Whether a plug-in is windowed or windowless depends on how you define it.

The way plug-ins are displayed on the web page is determined by the HTML tag that invokes them. This is up to the content developer or web page author. Depending on the tag and its attributes, a plug-in can be visible or hidden, or can appear as part of a page or as a full page in its own window. A web page can display a windowed or windowless plug-in in any HTML display mode; however, the plug-in must be visible for its window type to be meaningful. For information about the way HTML determines plug-in display mode, see "Using HTML to Display Plug-ins."

The Default Plug-in

When a specific plug-in is not registered to handle the media referred to in the HTML, Netscape Gecko invokes the *default plug-in* to help users find and install the right plug-in for that MIME type.

The blue puzzle piece that appears in the HTML page's plug-in window when the default plug-in loads is meant to signify that the browser is missing a piece that it needs to display or play the requested media.

How the plug-in HTML tag was coded determines what action is taken when the user clicks the plug-in piece. If the browser cannot handle the given MIME type, then the default plug-in checks to see if there is a plug-in referenced in the `OBJECT` tag that defines the media. If there is, then thedefault plug-in prompts the user to download that plug-in from the specified location. If a plug-in is not specified in the `OBJECT` tag, then the default plug-in looks for child elements, such as other `OBJECT` tag, which will provide more specific information about how to handle the specified media type.

Using HTML to Display Plug-ins

When a user browses to a web page that invokes a plug-in, how the plug-in appears (or does not appear) depends on two factors:

- The way the developer writes the plug-in determines whether it is displayed in its own window or is windowless.
- The way the content provider uses HTML tags to invoke the plug-in determines its display mode: whether it is embedded in a page, is part of a section, appears on its own separate page, or is hidden.

This section discusses using HTML tags and display modes. For information about windowed and windowless operation, see **Windowed and Windowless Plug-ins**.

For a description of each plug-in display mode, and which HTML tag to use to achieve it, go on to "**Plug-in Display Modes**." For details about the HTML tags and their attributes, go on to:

- "**Using the OBJECT Tag for Plug-in Display**"
- "**Using the EMBED Tag for Plug-in Display**"

Plug-in Display Modes

Whether you are writing an HTML page to display a plug-in or developing a plug-in for an HTML author to include in a page, you need to understand how the display mode affects the way plug-ins appear.

A plug-in, whether it is windowed or windowless, can have one of these display modes:

- embedded in a web page and visible
- embedded in a web page and hidden
- displayed as a full page in its own window

An **embedded plug-in** is part of a larger HTML document and is loaded at the time the document is displayed. The plug-in is visible as a rectangular subpart of the page (unless it is hidden). Embedded plug-ins are commonly used for multimedia images relating to text in the page, such as the Macromedia Shockwave plug-in. When Netscape Gecko encounters the `OBJECT` or `EMBED` tag in a document, it attempts to find and display the file represented by the `DATA` and `SRC` attributes, respectively. The `HEIGHT` and `WIDTH` attributes of the `OBJECT` tag determine the size of the embedded plug-in in the HTML page. For example, this `OBJECT` tag calls a plug-in that displays video:

```
<OBJECT DATA="newave.avi" TYPE=”video/avi”
  WIDTH=320
  HEIGHT=200
  AUTOSTART=true LOOP=true>
```

A **hidden plug-in** is a type of embedded plug-in that is not drawn on the screen when it is invoked. It is created by using the `HIDDEN` attribute of the `EMBED` tag. Here's an example:

```
<EMBED SRC="audiplay.aiff" TYPE="audio/x-aiff"
  HIDDEN=”true”>
```

NOTE: Whether a plug-in is windowed or windowless is not meaningful if the plug-in is invoked with the `HIDDEN` attribute.

You can also create hidden plug-ins using the `OBJECT` tag. Though the `OBJECT` tag has no `HIDDEN` attribute, you can create CSS rules to override the sizing attributes of the `OBJECT` tag

```
object {
    visibility: visible;
  }
object.hiddenObject {
    visibility: hidden ! important;
    width: 0px ! important;
    height: 0px ! important;
    margin: 0px ! important;
    padding: 0px ! important;
    border-style: none ! important;
    border-width: 0px ! important;
    max-width: 0px ! important;
    max-height: 0px ! important;
  }
```

In this case, the `OBJECT` tag that picks up these special style definitions would have a class of hidden. Using the `class` attribute and the CSS block above, you can simulate the behavior of the hidden plug-in in the `EMBED` tag:

```
<OBJECT DATA="audiplay.aiff" TYPE="audio/x-aiff"
  CLASS=”hiddenObject”>
```

A **full-page plug-in** is a visible plug-in that is not part of an HTML page. The server looks for the media (MIME) type registered by a plug-in, based on the file extension, and starts sending the file to the browser. Netscape Gecko looks up the MIME type and loads the appropriate plug-in if it finds a plug-in registered to that type. This type of plug-in completely fills the web page. Full-page plug-ins are commonly used for document viewers, such as Adobe Acrobat.

NOTE: The browser does not display scroll bars automatically for a full-page plug-in. The plug-in must draw its own scroll bars if it requires them.

The browser user interface remains relatively constant regardless of which type of plug-in is displayed. The part of the application window that does not display plug-in data does not change. The basic operations of the browser, such as navigation, history, and opening files, apply to all pages, regardless of the plug-ins in use.

Using the OBJECT Tag for Plug-in Display

The `OBJECT` tag is part of the HTML specification for generic inclusion of special media in a web page. It embeds a variety of object types in an HTML page, including plug-ins, Java components, ActiveX controls, applets, and images. `OBJECT` tag attributes determine the type of object to embed, the type and location of the object's implementation (code), and the type and implementation of the object's data.

Plug-ins were originally designed to work with the `EMBED` tag rather than the `OBJECT` tag (see "**Using the EMBED Tag for Plug-in Display**"), but the `OBJECT` tag itself provides some flexibility here. In particular, the `OBJECT` tag allows you to invoke another object if the browser cannot support the object invoked by the tag. The `EMBED` tag, which is also used for plug-ins, does not.

The `OBJECT` tag is also a part of the HTML W3C standard, for which see:

http://www.w3c.org/MarkUp/

Also, unlike the `APPLET` tag, `OBJECT` can contain other `HTML` attributes, including other `OBJECT` tags, nested between its opening and closing angle brackets. So, for example, though Netscape Gecko does not support the `CLASSID` attribute of the `OBJECT` tag—which was used for Java classes and ActiveX plug-ins embedded in pages—`OBJECT` tags can be nested to support different plug-in implementations.

See the Mozilla ActiveX project page in the "**Plug-in References**" section below for more information about embedding ActiveX controls in plug-ins or embedding plug-ins in ActiveX applications.

The following examples demonstrate this use of nested `OBJECT` tags with markup more congenial to Netscape Gecko included as children of the parent `OBJECT` tag.

Example 1: Nesting OBJECT Tags

```
<html>
<head>
<base href="http://www.macromedia.com/software/flash/">
<style>
  .myPlugin {
     width:  470px;
     height: 231px;
  }
</style>
<body>

<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
        codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=5,0,30,0"
        class="myPlugin">
  <param name=movie value="/software/flash/home_movie.swf">
  <param name=quality value=high>
  <param name="salign" value="tl">
  <param name="menu" value="0">

       <OBJECT data="/software/flash/home_movie.swf"
               type="application/x-shockwave-flash"
               class="myPlugin">
         <param name=quality value=high>
         <param name="salign" value="tl">
         <param name="menu" value="0">

              <object type="*" class="myPlugin">
               <param name="pluginspage" value="http://www.macromedia.com/
shockwave/download/index.cgi?P1_Prod_Version=ShockwaveFlash">
              </object>
       </object>
</object>


</body></html>
```

The outermost `OBJECT` tag defines the `CLASSID`; the first nested `OBJECT` uses the `TYPE` value "`application/x-shockwave-flash`" to load the shockwave plug-in, and the innermost `OBJECT` exposes a download page for users that do not already have the necessary plug-in. This nesting is quite common in the use of `OBJECT` tags, and lets you avoid code forking for different browser.

Nesting Rules for HTML Elements

The rules for descending into nested `OBJECT` and `EMBED` tags are as follows:

- The browser looks at the MIME type of the top element. If it knows how to deal with that MIME type (i.e., by loading a plug-in that's been registered for it), then it does so.
- If the browser cannot handle the MIME type, it looks in the element for a pointer to a plug-in that can be used to handle that MIME type. The browser downloads the requested plug-in.
- If the MIME type is unknown and there is no reference to a plug-in that can be used, the browser descends into the child element, where these rules for handling MIME types are repeated.

The rest of this section is a brief introduction to this HTML tag. For more information on the `OBJECT` tag and other tags used for plug-in display, see:

- W3C HTML 4.0 specification.

To embed a variety of object types in an HTML page, use the `OBJECT` tag.

```
<OBJECT
  CLASSID="classFile"
  DATA="dataLocation"
  CODEBASE="classFileDir"
  TYPE="MIMEtype"
  ALIGN="alignment"
  HEIGHT="pixHeight"
  WIDTH="pixWidth"
  ID="name"
>
```

The first set of `OBJECT` tag attributes are URLs.

- `CLASSID` is the `URL` of the specific object implementation. This attribute is similar to the `CODE` attribute of the `APPLET` tag. Though Netscape Gecko does not support this `OBJECT` attribute, you can nest `OBJECT` tags with different attributes to use the `OBJECT` tag for embedding plug-ins on any browser platform (see the example above).

- DATA represents the URL of the object's data; this is equivalent to the SRC attribute of EMBED.
- CODEBASE represents the URL of the plug-in; this is the same as the CODEBASE attribute of the APPLET tag. For plug-ins, CODEBASE is the same as PLUGINSPAGE.
- TYPE represents the MIME type of the plug-in; this is the same as the TYPE attribute of EMBED.
- HEIGHT, WIDTH, ALIGN are basic IMG/EMBED/APPLET attributes supported by OBJECT. HEIGHT and WIDTH are required for OBJECT tags that resolve to EMBED tags.
- Use the ID attribute, which specifies the name of the plug-in, if the plug-in is communicating with JavaScript. This is equivalent to the NAME attribute of APPLET and EMBED. It must be unique.

Using the Appropriate Attributes

It's up to you to provide enough attributes and to make sure that they do not conflict; for example, the values of WIDTH and HEIGHT may be wrong for the plug-in. Otherwise, the plug-in cannot be embedded.

Netscape Gecko interprets the attributes as follows: When the browser encounters an OBJECT tag, it goes through the tag attributes, ignoring or parsing as appropriate. It analyzes the attributes to determine the object type, then determines whether the browser can handle the type.

- If the browser can handle the type—that is, if a plug-in exists for that type—then all tags and attributes up to the closing </OBJECT> tag, except PARAM tags and other OBJECT tags, are filtered.
- If the browser cannot handle the type, or cannot determine the type, it cannot embed the object. Subsequent HTML is parsed as normal.

Using the EMBED Tag for Plug-in Display

A plug-in runs in an HTML page in a browser window. The HTML author uses the HTML EMBED tag to invoke the plug-in and control its display. Though the OBJECT tag is the preferred way to invoke plug-ins (see "**Using the OBJECT Tag for Plug-in Display**"), the EMBED tag can be used for backward compatibility with Netscape 4.x

browsers, and in cases where you specifically want to prompt the user to install a plug-in, because the default plug-in is only automatically invoked when you use the EMBED tag.

Netscape Gecko loads an embedded plug-in when the user displays an HTML page that contains an embedded object whose MIME type is registered by a plug-in. Plug-ins are embedded in much the same way as GIF or JPEG images are, except that a plug-in can be live and respond to user events, such as mouse clicks.

The EMBED tag has the following syntax and attributes:

```
<EMBED
  SRC="location"
  TYPE="MIMEtype"
  PLUGINSPAGE="instrURL"
  PLUGINURL="pluginURL"
  ALIGN="LEFT"|"RIGHT"|"TOP"|"BOTTOM"
  BORDER="borderWidth"
  FRAMEBORDER="NO"
  HEIGHT="height"
  WIDTH="width"
  UNITS="units"
  HIDDEN="TRUE|FALSE"
  HSPACE="horizMargin"
  VSPACE="vertMargin"
  NAME="pluginName"
  PALETTE="FOREGROUND"|"BACKGROUND"
>
...
</EMBED>
```

You must include either the SRC attribute or the TYPE attribute in an EMBED tag. If you do not, then there is no way of determing the media type, and so no plug-in loads.

The SRC attribute is the URL of the file to run. The TYPE attribute specifies the MIME type of the plug-in needed to run the file. Navigator uses either the value of the TYPE attribute or the suffix of the filename given as the source to determine which plug-in to use.

Use TYPE to specify the media type or MIME type necessary to display the plug-in. It is good practice to include the MIME type in all the plug-in HTML tags. You can use TYPE for a plug-in that requires no data, for example, a plug-in that draws an analog clock or fetches all of its data dynamically. For a visible plug-in, you must include WIDTH and HEIGHT if you use TYPE; no default value is used.

The PLUGINURL attribute is the URL of the plug-in or of the XPI in which the plug-in is stored (see "**Installing Plug-ins**" for more information on the XPI file format).

The EMBED tag has a number of attributes that determine the appearance and size of the plug-in instance, including these:

- The BORDER and FRAMEBORDER attributes specify the size of a border for the plug-in or draw a borderless plug-in
- HEIGHT, WIDTH, and UNITS determine the size of the plug-in in the HTML page. If the plug-in is not hidden, the HEIGHT and WIDTH attributes are required.
- HSPACE and VSPACE create a margin of the specified width, in pixels, around the plug-in.
- ALIGN specifies the alignment for the plug-in relative to the web page.

Use the HIDDEN attribute if you do not want the plug-in to be visible. In this case, you do not need the attributes that describe plug-in appearance. In fact, HIDDEN overrides those attributes if they are present.

Use the NAME attribute, which specifies the name of the plug-in or plug-in instance, if the plug-in is communicating with JavaScript.

For example, this EMBED tag loads a picture with the imaginary data type dgs.

```
<EMBED SRC="mypic.dgs" WIDTH=320 HEIGHT=200 BORDER=25
   ALIGN=right>
```

Netscape Gecko interprets the attributes as follows:

- SRC: Load the data file and determine the MIME type of the data.
- WIDTH and HEIGHT: Set the area of the page handled by the plug-in to 320 by 200 pixels. In general, use CSS to control the size and location of elements within an HTML page.
- BORDER: Draw a border 25 pixels wide around the plug-in.
- ALIGN: Align the plug-in at the right side of the web page.

The following example shows an EMBED tag nested within an OBJECT tag, which latter is necessary for browsers that do not support the EMBED tag.

Example 2: EMBED within OBJECT

```
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
   codebase="http://download.macromedia.com/pub/shockwave/
     cabs/flash/swflash.cab#version=5,0,30,0"
   width="749"
   height="68">

 <param name=movie value="/uber/nav/global_home.swf">
 <param name=quality value=high>
 <param name="BGCOLOR" value="#EEEEEE">
 <param name="salign" value="tl">
 <param name="menu" value="0">

 <embed src="/uber/nav/global_home.swf"
   quality=high pluginspage="http://www.macromedia.com/shockwave/
     download/index.cgi?P1_Prod_Version=ShockwaveFlash"
   type="application/x-shockwave-flash"
   width="749"
   height="68"
   bgcolor="#EEEEEE"
   salign="tl"
   menu="0">
 </embed>
</object>
```

Using Custom EMBED Attributes

In addition to these standard attributes, you can create private, plug-in-specific attributes and use them in the `EMBED` attribute to pass extra information between the HTML page and the plug-in code. The browser ignores these nonstandard attributes when parsing the HTML, but it passes all attributes to the plug-in, allowing the plug-in to examine the list for any private attributes that could modify its behavior.

For example, a plug-in that displays video could have private attributes that determine whether to start the plug-in automatically or loop the video automatically on playback, as in the following `EMBED` tag:

```
<EMBED SRC="myavi.avi" WIDTH=100 HEIGHT=125
   AUTOSTART=true LOOP=true>
```

With this `EMBED` tag, Netscape Gecko passes the values to the plug-in, using the arg parameters of the `NPP_New` call that creates the plug-in instance.

```
argc = 5
argn = {"SRC", "WIDTH", "HEIGHT", "AUTOSTART", "LOOP"}
argv = {"movie.avi", "100", "125", "TRUE", "TRUE"}
```

Netscape Gecko interprets the attributes as follows:

- `SRC`: Load the data file and determine the MIME type of the data.
- `WIDTH` and `HEIGHT`: Set the area of the page handled by the plug-in to 100 by 125 pixels.
- `AUTOSTART` and `LOOP`: Ignore these private attributes and pass them along to the plug-in with the rest of the attributes.

The plug-in must scan its list of attributes to determine whether it should automatically start the video and loop it on playback. Note that with an `OBJECT` tag, `PARAM` values are also sent in this array after the attributes, separated by a `PARAM` entry.

Plug-in References

- The Mozilla Plug-ins project page

  http://www.mozilla.org/projects/plugins/

- The Mozilla ActiveX Project

  http://www.iol.ie/~locka/mozilla/mozilla.htm

Plug-in Development Overview

Writing Plug-ins

Once you decide what you want your plug-in to do, creating it is a simple process. A basic overview of the plug-in development process is given in the following steps.

1. Plan your plug-in: decide on the services you want the plug-in software to provide and how it will interact with the browser and the special media for which the plug-in is created.

2. Decide the MIME type and file extension for the plug-in (see "**Registering Plug-ins**").

3. Set up your development environment properly. You can use a variety of environments to create a plug-in, but make sure that you have the necessary files from the mozilla source or from the plug-in SDK.

4. Create a plug-in project.

   You can either start from one of the samples provided for your operating system in the mozilla source directory, where plug-ins samples are already being built, or you can construct a new plug-in project in your own development environment using SDK-provided files. See the README in the plug-in SDK for more information about using the SDK and using the samples provided there.

5. Write your plug-in code and implement the appropriate Plug-in API methods for basic plug-in operation. You'll find an overview of the Plug-in API methods in this chapter, as well as separate chapters for all of the major functional areas of the Plug-in API. Also see "**Making Plug-ins Scriptable**" for more information about making plug-ins accessible from the browser.

6. Build the plug-in for your operating system. See "**Building Plug-ins**."

7. Install the plug-in in the plug-in directory for your operating system. See **Installing Plug-ins**.

8. Test your plug-in and debug as necessary.

9. Create an HTML page and embed the plug-in object. For information about the HTML tags to use, see "**Using HTML to Display Plug-ins**." To see your plug-in in action, simply display the HTML page that calls it in the browser.

Registering Plug-ins

Netscape Gecko identifies a plug-in by the MIME type it supports. When it needs to display data of a particular MIME type, the browser finds and invokes the plug-in object that supports that type. The data can come from either an `EMBED` tag in an HTML file (where the `OBJECT` or `EMBED` tag either specifies the MIME type directly or references a file of that type), from a separate non-HTML file of that MIME type, or from the server.

The server looks for the MIME type registered by a plug-in, based on the file extension, and starts sending the file to the browser. The browser looks up the media type, and if it finds a plug-in registered to that type, loads the plug-in software.

When it starts up, the browser checks for plug-in modules in the plug-in directory for the platform and registers them. It determines which plug-ins are installed and which types they support through a combination of user preferences that are private to the browser and the contents of the plug-ins directory.

A MIME type is made up of a major type (such as application or image) and a minor type, for example, image/jpeg . If you define a new MIME type for a plug-in, you must register it with IETF (Internet Engineering Task Force). Until your new MIME type is registered, preface its name with "x-", for example, image/x-nwim. For more information about MIME types, see these MIME RFCs:

- RFC 1521: "MIME: Mechanisms for Specifying and Describing the Forms of Internet Message Bodies"
- RFC 1590: "Media Type Registration Procedure."

There are some variations to how plug-ins are handled on different platforms. The following sections describe platform-specific discovery and registration:

- **Mac OS**
- **MS Windows**
- **Unix**

Mac OS

On the Mac OS platform, the Plug-ins folder is located in the same folder as the browser application. Plug-ins are identified by file type `NSPL`. When the browser starts up, it searches subfolders of the Plug-ins folder for plug-ins and follows aliases to folders and `NSPL` files. Plug-in filenames must begin with NP.

The MIME types supported by a plug-in are determined by its resources. 'STR#' 128 should contain a list of MIME types and file extensions in alternating strings. For example:

str 128	`MIME Type`
String 1	`video/quicktime`
String 2	`mov, moov`
String 3	`audio/aiff`
String 4	`aiff`
String 5	`image/jpeg`
String 6	`jpg`

Several other optional strings may contain useful information about the plug-in. Plug-ins must support 'STR#' 128 but are not required to support any of these others:

- `STR#' 127` can contain a list of MIME type descriptions corresponding to the types in 'STR#' 128 . For example, this description list corresponds to the types in the previous example: String 1: "QuickTime Video", String 4: "AIFF Audio", and String 5: "JPEG Image Format."
- `STR#' 126`: String 1 can contain a descriptive message about the plug-in. This message, which is in HTML format, is displayed by the browser in its "About Plug-ins" page. String 2 can contain the name of the plug-in, thus allowing the name the user sees to be different from the name of the file on disk.

MS Windows

On Windows, the plugins directory is located in the same directory as the browser application. Typical installations locate the plugins directory here:

```
C:\Program Files\Netscape\Netscape 6\Plugins
```

You can also find this directory through the Registry. The browser does not search subdirectories. Plug-ins must have a 8.3 filename beginning with NP and ending with .DLL.

The Windows version information for the plug-in DLL determines the MIME types, file extensions, file open template, plug-in name, and description. In the MIME types and file extensions strings, multiple types and extensions are separated by the "|" character, for example:

```
video/quicktime|audio/aiff|image/jpeg
```

For the browser to recognize the plug-in, the version stamp of the plug-in DLL must contain the following lines:

- File Extents: for file extensions
- MIME Type: for MIME types
- Language: for language in use

In your development environment, make sure your language is set to "US English" and the character set to "Windows Multilingual." The resource code for this language and character set combination is `040904E4`.

Unix

On Unix, the plugins directory is set by the environment variable `${MOZILLA_FIVE_HOME}/plugins`. Plug-in filenames must begin with `NP`.

To determine the MIME types of the plug-ins, the browser loads each plug-in library and calls its required `NPP_GetMIMEDescription` entry point. `NPP_GetMIMEDescription` should return a string containing the type, extension list, and type description separated by semicolons; for example, `image/xbm;xbm;X Bitmap`.

The browser also calls the plug-in's optional `NPP_GetValue` entry point to determine the plug-in name and description.

The calls to `NPP_GetMIMEDescription` and `NPP_GetValue` are made for registration purposes only. During registration, the browser does not call any other plug-in entry points, and the plug-in cannot call any other browser entry points at all.

Drawing a Plug-in Instance

Before drawing itself on the page, the plug-in must provide information about itself, set the window or other target in which it draws, arrange for redrawing, and handle events.

A windowless plug-in can call the following Netscape methods to draw itself:

- **NPN_ForceRedraw**: Force a paint message for windowless plug-ins.

- **NPN_InvalidateRect**: Invalidate an area in a windowless plug-in before repainting or refreshing.
- **NPN_InvalidateRegion**: Invalidate an area in a windowless plug-in before repainting or refreshing.

The browser calls these Plug-in methods:
- **NPP_GetValue**: Query the plug-in for information.
- **NPP_Print**: Request a platform-specific print operation for the instance.
- **NPP_SetValue**: Set the browser information.
- **NPP_SetWindow**: Set the window in which a plug-in draws.
- **NPP_HandleEvent**: Deliver a platform-specific event to the instance.

The plug-in can call these Netscape methods to query and set information:
- **NPN_GetValue**: Get the browser information.
- **NPN_SetValue**: Set plug-in the browser information.

For information about these processes, see "**Drawing and Event Handling**."

Handling Memory

Plug-in developers can take advantage of the memory features provided in the Plug-in API to allocate and free memory.
- Use the **NPN_MemAlloc** method to allocate memory from the browser.
- Use the **NPN_MemFree** method to free memory allocated with `NPN_MemAlloc`.
- Use the **NPN_MemFlush** method to free memory (Mac OS only) before calling memory-intensive Mac Toolbox calls.

Sending and Receiving Streams

Streams are objects that represent URLs and the data they contain. A stream is associated with a specific instance of a plug-in, but a plug-in can have more than one stream per instance. Streams can be produced by the browser and consumed by a plug-in instance, or produced by an instance and consumed by the browser. Each stream has an associated MIME type identifying the format of the data in the stream.

Streams produced by the browser can be automatically sent to the plug-in instance or requested by the plug-in. The plug-in can select one of these transmission modes:
- Normal mode: the browser sends the stream data sequentially to the plug-in as the data becomes available.
- Random-access mode: the browser allows the plug-in to request specific ranges of bytes from anywhere in the stream. This mode requires server support.

- File mode: the browser saves the data to a local file in cache and passes that file path to the plug-in.

Streams produced by the plug-in to send to the browser are like normal-mode streams produced by the browser, but in reverse. In the browser's normal-mode streams, the browser calls the plug-in to inform it that the stream was created and to push more data. In streams produced by the plug-in, by contrast, the plug-in calls Netscape functions to create a stream, push data into it, and delete it.

Working with URLs

The Plug-in API provides methods that plug-ins can use to retrieve data from or post data to a URL anywhere on the network, provide hyperlinks to other documents, post form data to CGI scripts using HTTP, or upload files to a remote server using FTP.

- Use **NPN_GetURL** to request the browser to load a URL into a particular browser window or frame for display, or to deliver the data of that URL to the plug-in instance in a new stream
- The **NPN_GetURLNotify** function operates like `NPN_GetURL`, except that it notifies the plug-in of the result when the operation completes.
- Use **NPN_PostURL** to send data to a URL from a memory buffer or file. The result from the server can also be sent to a particular browser window or frame for display, or delivered to the plug-in instance in a new stream.
- The **NPN_PostURLNotify** function operates like `NPN_PostURL`, except that it notifies the plug-in of the result when the operation completes.

For information about using these methods, see "**URLs**."

Getting Version and UI Information

The Netscape group of Plug-in API methods provides some basic services to the plug-in. You can use these Netscape methods:

- To identify the browser in which your plug-in is displayed: Use the NPN_UserAgent method to read this information.
- To determine whether plug-in and the browser versions are compatible and possibly provide alternative processing for different versions: Use the NPN_Version method to check for changes in major and minor version numbers.

For information about using these methods, see "**Version, UI, and Status Information**."

Displaying Messages on the Status Line

Functionally, your plug-in is seamlessly integrated into the browser and operates as an addition to current browser capabilities. To make the user feel that the plug-in is part of the the browser user interface, your plug-in can emulate the browser behavior by providing status line messages. Use the `NPN_Status` method to display a message on the status line.

For information about using this method, see "**Version, UI, and Status Information**."

Making Plug-ins Scriptable

Scriptable plug-ins are plug-ins that have that have been extended to provide methods that can be called from JavaScript and the DOM when accessed through the `OBJECT` or `EMBED` tag. Consider the following example, where a media player plug-in can be controlled with an `AdvanceToNextSong()` method called inside the `SCRIPT` tag:

```
<OBJECT id="myPlugin"
  type="audio/wav"
  data="music.wav"></OBJECT>
<SCRIPT>
  var thePlugin = document.getElementById('myPlugin');
  if (thePlugin)
    thePlugin.AdvanceToNextSong();
  else
    alert("Plugin not installed correctly");
</SCRIPT>
```

LiveConnect provided this sort of behavior for 4.x NPAPI plug-ins, but Netscape Gecko plug-ins now use XPConnect. Plug-ins that formerly used LiveConnect to make themselves scriptable in 4.x Netscape browsers have lost this possibility in the new XPCOM architecture upon which Netscape Gecko-based browsers are based. This is because there is no longer a guarantee of Java compatibility on a binary level due to the JRI/JNI switch. Plug-ins now use a mechanism called XPConnect to expose Netscape Communicator 4.x plug-ins to JavaScript in the browser interface.

Where LiveConnect was a bridge between Java and JavaScript, XPCOM is a more general framework for making components scriptable from the browser. In order to make plug-ins accessible via XPConnect, however, some changes have been made to the Mozilla code, and there are also some modifications you will have to make to your plug-in code.

For more information about XPCOM and XPConnect, see:

- www.mozilla.org/scriptable
- www.mozilla.org/projects/xpcom

Modifications to Your Plug-in Code

1. A unique interface ID should be obtained. The windows command `uuidgen` can generate this ID for you, as can `guidgen` on Unix.
2. An Interface Definition (.idl) file describing the plug-in scriptable interface should be added to the project (see **Example 1. Sample .idl file**).
3. A Scriptable instance object should be implemented in the plug-in. This class will contain native methods callable from JavaScript. This class should also inherit from `nsIClassInfo` and implement its methods to be able to request all necessary privileges from the Mozilla security manager (see **Example 2. Scriptable instance class**).
4. Cases should be added to the plug-in implementation of `NPP_GetValue` for two new scriptability additions to the NPPVariable enumeration type:

   ```
   NPPVpluginScriptableInstance = 10,
   NPPVpluginScriptableIID      = 11
   ```

   These two represent the scriptable plug-in instance and the unique ID of that plug-in, respectively. See "**Example 3. NPP_GetValue implementation**" for information about how to use these new enumerations in your code.

How to call plug-in native methods

The following examples demonstrate how easily the native methods of a plug-in can be called from JavaScript:

```
<embed type="application/plugin-mimetype">
<script language="javascript">
  var embed = document.embeds[0];
  embed.nativeMethod();
</script>
```

```
<object id=”plug” type="application/plugin-mimetype">
<script language="javascript">
  var object = document.getElementById(“plug”);
  object.nativeMethod();
</script>
```

Note that both of the ways to access the plug-in object—with the `embeds` array and with the `getElementById()` method—will work with the `EMBED` and `OBJECT` tags. The `embeds` property is used to return an array of embedded objects, which can then be indexed and used to call the method defined in the plug-in instance. The `document.getElementById()` returns a reference to an object specified by unique ID.

How to call JavaScript from the plug-in

When your plug-in is a scriptable component, it can be called from JavaScript in the interface, as the example above demonstrates. Note that you can also call JavaScript from your plug-in using some special methods described in a separate article:

http://www.mozilla.org/projects/plugins/scripting-plugins.html

This additional mechanism allows the plug-in to use JavaScript and access the DOM in the same way as other JavaScript objects in the interface:

```
<SCRIPT>
  var plugin = document.embeds[0];
  // tell the plugin the URL of this document.
  plugin.location = document.location;

  // read back the document's location
  alert('location  = ' + plugin.location);
</SCRIPT>
```

Scriptable Plug-in Lifetime

Scriptable plug-ins are not immediately unloaded from memory and scripting methods may still be called after the last plug-in instance is destroyed, since somebody may still hold on to the scriptable object. Instead, plug-ins are held in memory for a brief period of time so that the unloading can proceed safely after all objects have been released.

Scriptable plug-in building and installation overview

Though you do not need to have a copy of the Mozilla source tree in order to build your plug-in, making the plug-in interface scriptable will require Mozilla headers and the XPCOM compatible idl compiler, xpidl.exe. Note that you cannot use the MS DevStudio MIDL compiler for this. The header files and other supporting files you need are included in the Plug-in SDK.

This section provides a brief overview of the building and installation stages of your plug-in development. The following two sections, **Building Plug-ins** and **Installing Plug-ins** provide more detail about these important plug-in development steps.

The following steps describe how to build and install a plug-in called “TestPlugin”:

1. Compile `nsITestPlugin.idl` with the xpidl compiler. This will generate `nsITestPlugin.h` and `nsITestPlugin.xpt` files.
2. Put nsITestPlugin.xpt in the browser’s Plug-ins folder.
3. Build nptestplugin.dll with nsITestPlugin.h included for compiling scriptable instance class implementaion.

4. Put nptestplugin.dll in the Plug-ins folder.

Note that the “installation process” described here is a manual one, and merely describes how to get the browser to see and register your plug-in for the appropriate media type. See “**Installing Plug-ins**” for information on how to create a plug-in installation. Also see the following section, “**Building Plug-ins**”, for a more detailed account of the building process.

Example 1. Sample .idl file

```
#include "nsISupports.idl"

[scriptable, uuid(bedb0778-2ee0-11d5-9cf8-0060b0fbd8ac)]
interface nsITestPlugin : nsISupports {
      void nativeMethod();
};
```

Example 2. Scriptable instance class

```
#include "nsITestPlugin.h"
#include "nsIClassInfo.h"

// We must implement nsIClassInfo because it signals the
// Mozilla Security Manager to allow calls from JavaScript.

// helper class to implement all necessary nsIClassInfo method stubs
// and to set flags used by the security system
class nsClassInfoMixin : public nsIClassInfo
{
  // These flags are used by the DOM and security systems to signal that
  // JavaScript callers are allowed to call this object's scritable methods.
  NS_IMETHOD GetFlags(PRUint32 *aFlags)
    {*aFlags = nsIClassInfo::PLUGIN_OBJECT | nsIClassInfo::DOM_OBJECT;
     return NS_OK;}
```

```
  NS_IMETHOD GetImplementationLanguage(PRUint32 *aImplementationLanguage)
    {*aImplementationLanguage = nsIProgrammingLanguage::CPLUSPLUS;
     return NS_OK;}

  // The rest of the methods can safely return error codes...
  NS_IMETHOD GetInterfaces(PRUint32 *count, nsIID * **array)
    {return NS_ERROR_NOT_IMPLEMENTED;}
  NS_IMETHOD GetHelperForLanguage(PRUint32 language, nsISupports **_retval)
    {return NS_ERROR_NOT_IMPLEMENTED;}
  NS_IMETHOD GetContractID(char * *aContractID)
    {return NS_ERROR_NOT_IMPLEMENTED;}
  NS_IMETHOD GetClassDescription(char * *aClassDescription)
    {return NS_ERROR_NOT_IMPLEMENTED;}
  NS_IMETHOD GetClassID(nsCID * *aClassID)
    {return NS_ERROR_NOT_IMPLEMENTED;}
  NS_IMETHOD GetClassIDNoAlloc(nsCID *aClassIDNoAlloc)
    {return NS_ERROR_NOT_IMPLEMENTED;}
};

class nsScriptablePeer : public nsITestPlugin,
                         public nsClassInfoMixin
{
public:
    nsScriptablePeer();
    ~nsScriptablePeer();

    NS_DECL_ISUPPORTS
    NS_DECL_NSITESTPLUGIN
};

nsScriptablePeer::nsScriptablePeer()
{
    NS_INIT_ISUPPORTS();
}

nsScriptablePeer::~nsScriptablePeer()
```

```
{
}

// Notice that we expose our claim to implement nsIClassInfo.
NS_IMPL_ISUPPORTS2(nsScriptablePeer, nsITestPlugin, nsIClassInfo)

// the following method will be callable from JavaScript
NS_IMETHODIMP
nsScriptablePeer::NativeMethod()
{
    return NS_OK;
}
```

Example 3. NPP_GetValue implementation

The following example shows an implementation of `NPP_GetValue` with the updated parameters and a possible scenario of scriptable object life cycle.

```
#include "nsITestPlugin.h"

NPError NPP_New(NPMIMEType pluginType, NPP instance, uint16 mode,
                int16 argc, char* argn[], char* argv[], NPSavedData* saved)
{
    if(instance == NULL)
      return NPERR_INVALID_INSTANCE_ERROR;

    // just prime instance->pdata with null for the purpose of this example
    // it will be assigned to the scriptable interface later to keep its
    // association with the specific plugin instance
    instance->pdata = NULL;
    return rv;
}

NPError NPP_GetValue(NPP instance, NPPVariable variable, void *value)
{
```

```
    if(instance == NULL)
        return NPERR_INVALID_INSTANCE_ERROR;

    NPError rv = NPERR_NO_ERROR;
    static nsIID scriptableIID = NS_ITESTPLUGIN_IID;

    if (variable == NPPVpluginScriptableInstance) {

        // nsITestPlugin interface object should be associated with the plugin
        // instance itself. For the purpose of this example to keep things simple
        // we just assign it to instance->pdata after we create it.

        nsITestPlugin *scriptablePeer = (nsITestPlugin *)instance->pdata;

        // see if this is the first time and we haven't created it yet
        if (!scriptablePeer) {
            nsITestPlugin *scriptablePeer = new nsScriptablePeer();
            if (scriptablePeer)
                NS_ADDREF(scriptablePeer); // addref for ourself,
                                           // don't forget to release on
                                           // shutdown to trigger its destruction
        }
        // add reference for the caller requesting the object
        NS_ADDREF(scriptablePeer);
        *(nsISupports **)value = scriptablePeer;
    }
    else if (variable == NPPVpluginScriptableIID) {
        nsIID* ptr = (nsIID *)NPN_MemAlloc(sizeof(nsIID));
        *ptr = scriptableIID;
        *(nsIID **)value = ptr;
    }
    return rv;
}

NPError NPP_Destroy (NPP instance, NPSavedData** save)
{
```

```
    if(instance == NULL)
      return NPERR_INVALID_INSTANCE_ERROR;

    // release the scriptable object
    NS_IF_RELEASE(instance->pdata);
}
```

Building Plug-ins

Once you have added the special code and additional files to make your plug-in scriptable as described in the previous section, the build process is quite straightforward. In addition to the DLL that goes in the `plugins` folder, you must also place a type library and an extra header file in the appropriate places in your application directory. This section describes those extra scriptability steps in more detail.

Building, Platforms, and Compilers

Build resources have been supplied with the SDK for all of the major platforms. There are makefiles for the Unix platform, project files for the Windows and Macintosh IDEs, definition files, resources files, and other resources for building the samples in the SDK and your own plug-in projects. Netscape Gecko plug-ins can also be compiled by well-known compilers on all the major platforms—though using those compilers competently is of course outside the scope of this manual.

All the resources you need—the definition files, the source files, the resource files—can be found in the Plug-in SDK, which is available in the mozilla source tree and also as separately downloadable and buildable software kit. The basic plug-in example, located in the mozilla source at `mozilla/modules/plugin/tools/sdk/samples/basic`, has all the files you need to build a simple plug-in on the major platforms.

Building Carbonized Plug-ins for Mac OSX

The building process for Mac OSX plug-ins is very like that forMac “classic” plug-ins and plug-ins on other platforms. There are, however, a couple of differences you must be aware of if you are going to successfully compile your plug-in for the Mac OSX platform.

The main change is visible in the npupp.h header file, where the preprocessor variable _NPP_USE_UPP_ is set to FALSE or 0, because TARGET_API_MAC_CARBON is true:

```
/* NPP_Initialize */

#define _NPUPP_USE_UPP_ (TARGET_RT_MAC_CFM && !TARGET_API_MAC_CARBON)

#if _NPUPP_USE_UPP_
typedef UniversalProcPtr NPP_InitializeUPP;

enum {
   uppNPP_InitializeProcInfo = kThinkCStackBased
      | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(0))
      | RESULT_SIZE(SIZE_CODE(0))
};

#define NewNPP_InitializeProc(FUNC)\
     (NPP_InitializeUPP) NewRoutineDescriptor((ProcPtr)(FUNC),
uppNPP_InitializeProcInfo, GetCurrentArchitecture())
#define CallNPP_InitializeProc(FUNC)\
     (void)CallUniversalProc((UniversalProcPtr)(FUNC),
uppNPP_InitializeProcInfo)

#else

typedef void (* NP_LOADDS NPP_InitializeUPP)(void);
#define NewNPP_InitializeProc(FUNC)\
     ((NPP_InitializeUPP) (FUNC))
#define CallNPP_InitializeProc(FUNC)\
     (*(FUNC))()

#endif
```

When this is the case, all of the function pointers in the `NPPluginFuncs` struct, also described in the npupp.h header file, will be actual function pointers and not “routine descriptors,” which aren’t supported in the Carbon runtime:

```
typedef struct _NPPluginFuncs {
    uint16 size;
    uint16 version;
    NPP_NewUPP newp;
    NPP_DestroyUPP destroy;
    NPP_SetWindowUPP setwindow;
    NPP_NewStreamUPP newstream;
    NPP_DestroyStreamUPP destroystream;
    NPP_StreamAsFileUPP asfile;
    NPP_WriteReadyUPP writeready;
    NPP_WriteUPP write;
    NPP_PrintUPP print;
    NPP_HandleEventUPP event;
    NPP_URLNotifyUPP urlnotify;
    JRIGlobalRef javaClass;
    NPP_GetValueUPP getvalue;
    NPP_SetValueUPP setvalue;
} NPPluginFuncs;
```

Finally, in the Mac Classic plug-ins, the main entry point is required to be an exported symbol called “mainRD”, which is a routine descriptor for the plug-ins main function:

```
#ifdef XP_MAC
/
************************************************************************************
 * Mac platform-specific plugin glue stuff

************************************************************************************
*/


/*
 * Main entry point of the plugin.
 * This routine will be called when the plugin is loaded. The function
 * tables are passed in and the plugin fills in the NPPluginFuncs table
```

```
 * and NPPShutdownUPP for Netscape's use.
 */

#if _NPUPP_USE_UPP_

typedef UniversalProcPtr NPP_MainEntryUPP;
enum {
   uppNPP_MainEntryProcInfo = kThinkCStackBased
      | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(NPNetscapeFuncs*)))
      | STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(NPPluginFuncs*)))
      | STACK_ROUTINE_PARAMETER(3, SIZE_CODE(sizeof(NPP_ShutdownUPP*)))
      | RESULT_SIZE(SIZE_CODE(sizeof(NPError)))
};
#define NewNPP_MainEntryProc(FUNC)\
      (NPP_MainEntryUPP) NewRoutineDescriptor((ProcPtr)(FUNC),
uppNPP_MainEntryProcInfo, GetCurrentArchitecture())
#define CallNPP_MainEntryProc(FUNC,  netscapeFunc, pluginFunc, shutdownUPP)\
      CallUniversalProc((UniversalProcPtr)(FUNC),
(ProcInfoType)uppNPP_MainEntryProcInfo, (netscapeFunc),
(pluginFunc), (shutdownUPP))
```

However, in the Carbon runtime plug-ins, it’s good form if the plug-in exports a “main” entry point, which is expected to have the same prototype. At a bare minimum, the shared library’s “main” entry point must be set to such a routine.

Getting and Using the xpidl Compiler

The xpidl compiler that you must use to create the type library and the header file for your plug-in is a regular product of the mozilla build process. In the bin directory of your mozilla build, you ought to see the xpidl binary. Use the -m option to specify which kind of output you want, as in the following usage note.

```
Usage: ./xpidl [-m mode] [-w] [-v]
          [-I path] [-o basename | -e filename.ext] filename.idl
       -a emit annotations to typelib
       -w turn on warnings (recommended)
       -v verbose mode (NYI)
       -I add entry to start of include path for ``#include "nsIThing.idl"''
       -o use basename (e.g. ``/tmp/nsIThing'') for output
       -e use explicit output filename
       -m specify output mode:
          header        Generate C++ header            (.h)
          typelib       Generate XPConnect typelib     (.xpt)
          doc           Generate HTML documentation    (.html)
          java          Generate Java interface        (.java)
```

For example, to create a header file for a plug-in IDL file nsITestPlugin.idl, you would type the following at the command prompt:

```
./xpidl -m header nsITestPlugin.idl
```

The resulting header file, nsITestPlugin.h, should then be included when the nsTestPlug.dll is built.

Type Libraries

In addition to the header file, you must also create a *type library* file for your plug-in. This file—in our example, nsITestPlugin.xpt—can also easily be generated from the xpidl compiler, and should be placed in the Plugins subdirectory of the browser application.

The type library is a special binary independent interface file that exposes the interface(s) of an object in a way that allows them to be used uniformly across platforms, languages, and programming environments. The type library provides the information about the interface at run-time, which is required in a cross-platform component framework like XPCOM.

To create a type library file for the nsITestPlugin.idl IDL, you would type the following at the command prompt:

```
./xpidl -m typelib nsITestPlugin.idl
```

Installing Plug-ins

With the redesign of the Netscape and Mozilla browsers, there has been a dramatic change to the way that plug-ins and other software are installed. Netscape Gecko now provides a cross-platform installation API that you can use to install new browser components, plug-ins, applications, or any other software.

This API can be used in one of two ways. You can create a small installation script to download and execute a binary installer for the plug-in, as described in the **Native Installers** below. Or you can do the entire installation using the XPInstall API, which is documented in the **XPI Plug-ins Installations** section below that.

For more general information about the API, see:

The XPInstall API Reference

Native Installers

Plug-ins must use the XPInstall API to install themselves in the appropriate area. They may also use other binary installers, as before, in which case the XPInstall archive and its installation script are effectively a small wrapper for the installer

executable, downloading that binary and executing it on the user's system. The following installation script example gives you some idea of how simple this "wrapper" can be.

```
// DJ Double-Decker Plug-in Installer
err    = initInstall("
   DJ Double-Decker Plug-in Installer",
   "DJDD",
   "0.9");

logComment("initInstall() returned: " + err);

err = execute("djdd.exe", "", true);
logComment("execute() returned: " + err);

if(!err)
{
  err = performInstall();
  logComment("performInstall() returned: " + err);
}
```

Even with the optional logging (i.e., the `logComment()` method used after each main step to check the return value of that function), the installation is less than ten lines.

Using an XPInstall script like this to wrap the installer has the additional advantage of running in the same process as the browser, which means that you can invoke the installer executable and hand back control immediately.

`initInstall` begins every installation script with parameters representing the name and other information about the installation. The next line uses the `execute()` method (which is a member of the Install object, implicit in installation script just as the window object is implicit in browser scripts) to execute the installer contained within the archive. `performInstall()` begins the actual installation. Note that you do not have to install the installer in order to execute it on the local system. See the XPInstall API for more information about cross-platform installations, and see the second example below for a more detailed plug-in installation, in which the XPInstall API performs all of the necessary steps to install the plug-in and its supporting files and register it with the browser.

This script is included in a special archive called a XPI. When a separate executable is performing the actual installation, the contents of that XPI may be nothing other than the installer executable and the install.js installation script.

XPI Plug-ins Installations

You can also use the XPInstall API do the installation yourself, without using a third-party installer. The following script works on any platform, and installs the JRE 1.3 plug-in the JRE in the Netscape 6 browser. This sort of script can easily be adapted to install any type of plug-in.

```
// this function verifies disk space in kilobytes
function verifyDiskSpace(dirPath, spaceRequired)
{
  var spaceAvailable;

  // Get the available disk space on the given path
  spaceAvailable = fileGetDiskSpaceAvailable(dirPath);

  // Convert the available disk space into kilobytes
  spaceAvailable = parseInt(spaceAvailable / 1024);

  // do the verification
  if(spaceAvailable < spaceRequired)
  {
    logComment("Insufficient disk space: " + dirPath);
    logComment("  required : " + spaceRequired + " K");
    logComment("  available: " + spaceAvailable + " K");
    return(false);
  }

  return(true);
}


var srDest = 38628;
```

```
var err = initInstall("Sun Java 2", "/Sun/Java2", "1.3");
logComment("initInstall: " + err);

var fPlugins= getFolder("Plugins");
logComment("plugins folder: " + fPlugins);

if (verifyDiskSpace(fPlugins, srDest))
{
    err = addDirectory("JRE_Plugin_Linux_i386",
                       "1.3",
                       "jre-image-i386",   // jar source folder
                       fPlugins,           // target folder
                       "java2",            // target subdir
                       true );             // force flag

    logComment("addDirectory() returned: " + err);

    // create symlink: plugins/libjavaplugin_oji.so ->
    //                 plugins/java2/plugin/i386/libjavaplugin_oji.so
    var lnk = fPlugins + "libjavaplugin_oji.so";
    var tgt = fPlugins + "java2/plugin/i386/ns600/libjavaplugin_oji.so";
    var ignoreErr = execute("symlink.sh", tgt + " " + lnk, true);
    logComment("execute symlink.sh "+tgt+" "+lnk+" returned: "+ignoreErr);

    if (err==SUCCESS)
    {
       err = performInstall();
       logComment("performInstall() returned: " + err);
    }
    else
    {
       cancelInstall(err);
       logComment("cancelInstall() returned: " + err);
    }
}
else
```

```
cancelInstall(INSUFFICIENT_DISK_SPACE);
```

Note that this script installs the Linux JRE plug-in and assumes you are running Linux, but you can also use the XPInstall API to check the platform type, check for the presence of other files, and perform other preparatory functions in your installation scripts.

Also note the use of the "Plugins" keyword in the `getFolder()` function to locate and specify the plug-ins subdirectory in a cross-platform way. The returned object, `fPlugins`, is used as the target folder for installation of this binary file in the `addDirectory()` function that actually specifies where the files in the XPI are to be installed on the local machine.

Plug-in Installation and the Windows Registry

An important aspect of the installation process on the Windows platform is the reading of registry keys to determine how many Netscape Gecko-based browsers are installed locally, which they are, and how they are configured for plug-ins.

Whether you are using a native Windows installer like InstallShield or writing installation scripts using the XPInstall API (see "**XPI Plug-ins Installations**"), you can access the registry, read and write data about your plug-in, and customize your installation for the different Netscape Gecko installation targets, as this section describes.

The registry keys that affect the installation of plug-ins are subkeys of the various Netscape Gecko-based products enumerated under:

```
HKEY_LOCAL_MACHINE\Software\Mozilla
```

The products are listed as subkeys of the Mozilla key. You can enumerate these subkeys to get the Netscape Gecko-based browsers, and further enumerate those subkeys to read such important configuration information as where in the browser application directories the plug-in should be installed, which version is installed, and so on.

The Plugins key-value pair shows where plug-ins should be installed for that Netscape Gecko-based product:

```
Plugins = C:\Program Files\Netscape\Netscape 6\Plugins
```

For all but the newest Netscape Gecko-based products, the Components key-value pair also holds an important piece of information: As described in the "**Type Libraries**" section above, Netscape Gecko-based products require that you put the type library file, or XPT, in the `Components` subdirectory.

```
Components = C:\Program Files\Netscape\Netscape 6\Components
```

Also, the product subkey (e.g., Mozilla/Netscape 6 6.2.1) has a `bin` subkey which exposes the PathToExe key-value pair:

```
PathToExe = C:\Program Files\Netscape\Netscape 6\netscp6.exe
```

See the XPInstall registry manipulation example for more information about how these key values from the registry can be used to steer your installation for different targets.

If you are using a native installer, then that installer will have its own way to access and update the registry. If you are using the XPInstall API, then you can use the winReg function to find the plug-in subdirectories where your software should be installed, as the following example demonstrates.

```
var winreg = getWinRegistry();

winreg.setRootKey(winreg.HKEY_LOCAL_MACHINE);
var index = 0;
var baseKey = "Software\\Mozilla";

while ( (MozillaVersion = winreg.enumKeys(baseKey,index)) != null )
{
  logComment("MozillaVersion = " + MozillaVersion);
  subkey = baseKey + "\\" + MozillaVersion + "\\Extensions";
  pluginsDir = winreg.getValueString ( subkey, "Plugins" );
  if ( pluginsDir )
    logComment("pluginsDir = " +  pluginsDir);
  else
    logComment("No plugins dir for " + baseKey + "\\" + MozillaVersion);
  index++;
}
```

When combined with the installation examples above, this kind of parsing of the Windows registry can make it easy for you to install plug-ins on different platforms and browsers.

Initialization and Destruction

This chapter describes the methods that provide the basic processes of initialization, instance creation and destruction, and shutdown.

- **Initialization**: The browser calls the Plug-in API function **NP_Initialize** when the plug-in code is first loaded.
- **Instance Creation**: The browser calls the Plug-in API function **NPP_New** when the instance is created.
- Instance Destruction: The plug-in instance is deleted when the user leaves the instance page or closes the instance window; the browser calls the function **NPP_Destroy** to tell the plug-in that the instance is being deleted.
- Shutdown: When the last instance of a plug-in is deleted, the plug-in code is unloaded from memory and the browser calls the function **NP_Shutdown**. Plug-ins consume no resources, other than disk space, if not referenced.

This chapter ends with **Initialize and Shutdown Example**, which includes the **NP_Initialize** and **NP_Shutdown** methods.

Initialization

The browser calls **NP_Initialize** when a plug-in is loaded and before the first instance is created. Use this function to allocate the memory and resources shared by all instances of your plug-in.

```
NPError NP_Initialize(void){
};
```

After the last plug-in instance is deleted, the browser calls **NP_Shutdown**, which releases the memory or resources allocated by `NP_Initialize`. For an example that shows the use of both the `NP_Initialize` and `NP_Shutdown` methods, see **Initialize and Shutdown Example**

During initialization, when the browser encounters data of a MIME type registered for a plug-in (either embedded in an HTML page or in a separate file), it loads the plug-in code into memory (if it hasn't been loaded already) and creates a new instance of the plug-in. For more information, see **Registering Plug-ins**

Plug-ins are native code libraries: .DLL files on Windows, .SO or .DSO files on Unix, and PowerPC Shared Library files or 68K code resources on Mac OS. To reduce memory overhead, plug-ins are usually loaded only when needed and released as soon as possible.

In the initialization process, the browser passes the plug-in two tables of function pointers for all API calls:

- One table lists all API calls from the plug-in to the browser. This table is filled out by the browser before the initialization call.
- The other table lists all API calls from the browser to the plug-in. This table is filled out by the plug-in during the initialization call.

The function tables also contain version information that the plug-in checks to verify that it is compatible with the API capabilities provided by the application. To check this information, use **NPN_Version**.

No plug-in API calls can take place in either direction until the initialization completes successfully, with the exception of the functions `NP_Initialize` and `NP_Shutdown`, which are not in the function tables. However, because `NP_Initialize` is called at the end of the initialization process, you can call other methods, such as such as `NPP_MemAlloc` and `NPP_Status`, from `NP_Initialize`.

Instance Creation

After initialization, the plug-in instance is created. More than one instance of the same plug-in can exist if a single page contains multiple embedded plug-ins, or if several browser windows are open and display the same data type. At this point, a plug-in can call the **NPN_SetValue** function to specify whether it is windowed (the default) or windowless.

Plug-in instance are created with **NPP_New** and destroyed with **NPP_Destroy**. `NPP_New` informs the plug-in of the creation of a new instance with the specified MIME type. You can allocate instance-specific private data at this time.

```
NPError NPP_New(NPMIMEType pluginType,
NPP instance, uint16 mode,
int16 argc, char *argn[],
char *argv[], NPSavedData *saved);
```

The `pluginType` parameter represents the MIME type of this instance of the plug-in. You can assign more than one MIME type to a plug-in, which could potentially allow the plug-in to respond to data streams of different types with different interfaces and behavior.

The `instance` parameter represents an `NPP` object, created by the browser. You can store the instance-specific private data in its pdata field (instance->pdata).

The `mode` parameter identifies the display mode in which the plug-in was invoked, either `NP_EMBED` or `NP_FULL`.

- `NP_EMBED` means that the instance was created by an `EMBED` and shares the browser window with other content.
- `NP_FULL` means that the instance was created by a separate file and is the primary content in the window.

The next three parameters pass parameters from the `EMBED` tag that called the plug-in. The argc parameter is the number of HTML arguments in the tag. It determines the number of attributes in the arrays specified by the argn and argv parameters.

The arguments in the `EMBED` tag are name-value pairs made up of the attribute name (for example, `ALIGN`) and its value (for example, top). The argn array contains the attribute names; the argv array contains the attribute values.

The browser ignores any nonstandard attributes in an `EMBED` tag, so the web page author can use the arg parameters to specify private attributes defined for a particular plug-in. For example, the following `EMBED` tag has the standard attributes `SRC`, `HEIGHT`, and `WIDTH` and the private attribute `LOOP`:

```
<EMBED SRC="movie.avi" HEIGHT=100 WIDTH=100 LOOP=TRUE>
```

With the `EMBED` tag in the example, the browser passes the values in argv to the plug-in instance:

```
argc = 4
argn = { "SRC", "HEIGHT", "WIDTH", "LOOP" }
argv = { "movie.avi", "100", "100", "TRUE" }
```

The saved parameter allows an instance of a plug-in to save its data and, when the instance is destroyed, pass the data to the next instance of the plug-in at the same URL. The data is saved in the History list. As long as the plug-in still appears in this list, that saved data is associated with the page; any new instances receive this data.

Instance Destruction

Plug-in instances are created with **NPP_New** and destroyed with **NPP_Destroy**. The browser calls `NPP_Destroy` when a plug-in instance is deleted, usually because the user has left the page containing the instance, closed the window, or quit the application. If this is the last instance created by a plug-in, **NP_Shutdown** is called.

You should not perform any graphics operations in `NPP_Destroy` because the instance window is no longer guaranteed to be valid. Also, be sure to delete any private instance-specific information stored in the plug-in's instance->pdata.

```
NPError NPP_Destroy(NPP instance, NPSavedData **save);
```

The `instance` parameter represents the plug-in instance to delete.

The plug-in can use the optional `save` parameter to save data for reuse by a new instance with the same URL. The data is passed to `NPP_New` through its saved parameter. For example, a video player could save the last frame number to be displayed. When the user returns to the page, the previous frame number is passed to the new instance of the plug-in, so it can initially display the same frame.

Note that you cannot count on data being saved this way; the data may be lost if the browser restarts or purges memory. Ownership of the buf field of the `NPSavedData` structure passes from the plug-in to the browser when `NPP_Destroy` returns.

The example in this section sets up a buffer and allocates memory for it. You can use this type of buffer to handle data saved from one instance of a plug-in to another. The example shows the use of the optional save parameter of `NPP_Destroy` and saved parameter of `NPP_New`.

- In `NPP_New`, the saved parameter contains previously saved data for this instance of the plug-in (saved by `NPP_Destroy`). The plug-in must free the memory for `NPSavedData` and the buffer it contains.
- In `NPP_Destroy`, the save parameter specifies state or other information to save for reuse by a new instance with the same URL.

To ensure that the browser does not crash or leak memory when the saved data is discarded, the buf field should be a flat structure (a simple structure with no allocated substructures) allocated with `NPN_MemAlloc`, as in this example:

```
char* myData = "Here is some saved data.\n";
int32 myLength = strlen(myData) + 1;
*save = (NPSavedData*)
NPN_MemAlloc(sizeof(NPSavedData));
(*save)->len = myLength;
(*save)->buf = (void*) NPN_MemAlloc(myLength);
strcpy((*save)->buf, myData);
```

If you allocate saved instance data in `NPP_Destroy`, be sure to allocate memory with this function, since the browser can delete the saved data with the equivalent of `NPN_MemAlloc` at any time.

Shutdown

When the application no longer needs the plug-in, it is shut down and released. **NP_Shutdown** gives you an opportunity to delete data allocated in **NP_Initialize** to be shared by all instances of a plug-in. The browser calls the plug-in's `NP_Shutdown` function, which informs the plug-in that its library is about to be unloaded, and gives it a chance to cancel any outstanding I/O requests, delete threads it created, free any memory it allocated, and perform any other closing tasks.

The `NP_Shutdown` function releases memory or resources shared across all instances of a plug-in. It is called once after the last instance of the plug-in is destroyed, before releasing the plug-in library itself.

```
void NP_Shutdown(void);
```

For an example that shows both the `NP_Initialize` and `NPN_Shutdown` methods, see "**Initialize and Shutdown Example**."

Initialize and Shutdown Example

This example demonstrates the use of the **NP_Initialize** and **NP_Shutdown** methods.

```
/* Define global variable to hold the user agent string. */
static char* userAgent = NULL;

/* Initialize function. */
NPError NP_Initialize(void)
{
/* Get the user agent from the browser. */
char* result = NPN_UserAgent();
if (result == NULL) return NPERR_OUT_OF_MEMORY_ERROR;

/* Allocate some memory so that you can keep a copy of it. */
userAgent = (char*) NPN_MemAlloc(strlen(result) + 1);
if (userAgent == NULL) return NPERR_OUT_OF_MEMORY_ERROR;

/* Copy the string to your memory. */
strcpy(userAgent, result);
return NPERR_NO_ERROR;
}

/* Shutdown function */
NPError NP_Shutdown(void)
{
/* Delete the memory you allocated. */
if (userAgent != NULL)
NPN_MemFree(userAgent);
return NPERR_NO_ERROR;
}
```

Drawing and Event Handling

This chapter tells how to determine whether a plug-in instance is windowed or windowless, how to draw and redraw plug-ins, and how to handle plug-in events.

When it comes to determining the way a plug-in instance appears in a web page, you (and the web page author) have many options. The content provider who writes the web page determines its display mode : whether the plug-in is embedded, or displayed in its own separate page. You determine whether a plug-in is windowed or windowless by the way you define the plug-in itself.

- A windowed plug-in is drawn into its own native window (or portion of a native window) on a web page. A windowed plug-in is opaque, hiding the part of the page beneath its display window. This type of plug-in determines when it draws itself.
- A windowless plug-in does not require a native window. It is drawn in a target called a drawable , which corresponds to either the browser window or an off-screen bitmap. A drawable can be defined in several ways, depending on the platform. Windowless plug-ins can be opaque or transparent. A windowless plug-in draws itself only in response to a paint message from the browser.

For information about the way HTML determines plug-in display mode, see "**Using HTML to Display Plug-ins**."

- **The NPWindow Structure**
- **Drawing Plug-ins**
- **Windowed Plug-ins**
- **Windowless Plug-ins**

NOTE: Windowless plug-ins are currently not supported on the X Windows platform.

The NPWindow Structure

When a plug-in is loaded, it is drawn into a target area. This target is either the windowed plug-in's native window, or the drawable of a windowless plug-in. The NPWindow structure represents either the native window or a drawable. This structure contains information about coordinate position, size, the state of the plug-in (windowed or windowless), and some platform-specific information.

NOTE: When a plug-in is drawn to a window, the plug-in is responsible for preserving state information and ensuring that the original state is restored.

For windowless plug-ins, the browser calls the NPP_SetWindow method with an NPWindow structure that represents a drawable. For windowed plug-ins, the browser calls the NPP_SetWindow method with an NPWindow structure that represents a window.

The NPWindow Structure

```
typedef enum {
NPWindowTypeWindow = 1,
NPWindowTypeDrawable
} NPWindowType;
typedef struct _NPWindow
{
void*   window;   /* Platform-specific handle      */
uint32  x;        /* Position of top-left corner   */
uint32  y;        /*   relative to a Netscape page */
uint32  width;    /* Maximum window size           */
uint32  height;
NPRect  clipRect; /* Clipping rectangle in port
coordinates */
#ifdef XP_UNIX
void * ws_info; /* Platform-dependent additional data
*/
#endif /* XP_UNIX */
NPWindowType  type;   /* Whether this is a window or a
drawable */

} NPWindow;
```

The window parameter is a platform-specific handle to a native window element in the browser window hierarchy on Windows and Unix. On Mac OS, window is a pointer to an NP_Port.

The x and y fields specify the top-left corner of the plug-in relative to the page.

The width and height fields specify the dimensions of the plug-in area. These values should not be modified by the plug-in.

The `clipRect` field defines the clipping rectangle of the plug-in in a coordinate system where the origin is the top-left corner of the drawable or window. The browser calls `NPP_SetWindow` whenever the drawable changes.

The `type` field indicates the NPWindow type of the target area:

- `NPWindowTypeWindow`: Windowed plug-in. The window field holds a platform-specific handle to a window.
- `NPWindowTypeDrawable`: Windowless plug-in. The window field holds a platform-specific handle to a drawable, as follows:
    - Windows: `HDC`
    - Mac OS: pointer to `NP_Port` structure.

In both cases, the drawable can be an off-screen pixmap.

Drawing Plug-ins

This section describes the methods and processes you use in drawing both windowed and windowless plug-ins. Processes that apply to only one of these plug-in types are described in the following sections.

The plug-in uses these methods to draw plug-ins and to handle events:

Plug-in methods, called by the browser:

`NPP_HandleEvent`	Deliver a platform-specific event to the instance.
`NPP_Print`	Request a platform-specific print operation for the instance.
`NPP_SetWindow`	Set the window in which a plug-in draws.

Browser-side methods, called by the plug-in:

`NPN_ForceRedraw`	Force a paint message to a windowless plug-in.
`NPN_InvalidateRect`	Invalidate an area in a windowless plug-in before repainting or refreshing.
`NPN_InvalidateRegion`	Invalidate a region in a windowless plug-in before repainting or refreshing.

Printing the Plug-in

The browser calls the **NPP_Print** method to ask the plug-in instance to print itself.

```
void NPP_Print(NPP instance, NPPrint *printInfo);
```

The `instance` parameter represents the current plug-in.

The `PrintInfo` parameter determines the print mode. It is set to either `NP_FULL` to indicate full-page plug-in printing, or `NP_EMBED` if this is an embedded plug-in printed as part of the window in which it is embedded.

- An embedded plug-in shares printing with the browser. The plug-in prints the part of the page it occupies, and the browser handles the rest of the printing process, including displaying print dialog boxes, getting the printer device context, and, of course, printing the rest of the page.

  An embedded plug-in can set the `pluginPrinted` field in its `PrintInfo` parameter to false (the default). This is a field of the `_NPFullPrint` substructure of the `NPPrint` structure. The browser displays the necessary print dialog boxes and calls `NPP_Print` again. This time, `PrintInfo->mode` should be set to `NP_EMBED`.

- A full-page plug-in handles the print dialog boxes and printing process as it sees fit. In this case, before the browser displays any print dialog boxes, `NPP_Print` is called with `PrintInfo->mode` equal to `NP_FULL`. On Mac OS, full-page printing requires that the field `PrintInfo` contain a standard Mac OS `THPrint` (see Printing.h).

Of course, `NPP_Print` is also called with `PrintInfo->mode` equal to `NP_EMBED` when the instance is embedded. In this case, platformPrint->embedPrint.window contains the window in which the plug-in should print.

On MS Windows, note that the coordinates for the window rectangle are in `TWIPS` format. For this reason, you need to convert the x- and y-coordinates using the Windows API call `DPtoLP` when you output text.

Setting the Window

The browser calls the **NPP_SetWindow** function to set the window in which a plug-in draws or returns an error code. This window is valid for the life of the instance, or until `NPP_SetWindow` is called again with a different value.

Subsequent calls to `NPP_SetWindow` for a given instance usually mean that the window has been resized. If either window or window->window is null, the plug-in must not perform any additional graphics operations on the window and should free any associated resources.

```
NPError NPP_SetWindow(NPP instance, NPWindow *window);
```

The `instance` parameter represents the current plug-in.

The window parameter is a pointer to the drawing target for the plug-in. For windowless plug-ins, the platform-specific window information specified in window->window is a platform-specific handle to a drawable.

MS Windows and Unix

For windowed plug-ins on MS Windows and Unix, the window->window field is a handle to a subwindow of the Netscape window hierarchy.

Mac OS

The window->window field points to an `NP_Port` structure.

Getting Information

To receive information from the browser, the plug-in calls the **NPN_GetValue** method.

```
NPError NPN_GetValue(NPP instance,
NPNVariable variable, void *value);
```

The `instance` parameter represents the current plug-in.

Unix and MS Windows

The queried information is returned in the variable parameter. This parameter is valid only for the Unix and MS Windows platforms. For Unix, the values are either the current display (`NPNVxDisplay`) or the application's context (`NPNVxtAppContext`). For MS Windows, the value is the native window on which the plug-in drawing occurs (`NPNVnetscapeWindow`).

The value parameter contains the name of the plug-in.

You can also use `NPN_GetValue` to help create a menu or dialog box for a windowless plug-in.

Windowed Plug-ins

The browser gives each windowed plug-in its own native window, often a child window of the the browser window itself, to draw into. The plug-in has complete control over drawing and event handling within that window.

On Mac OS , the browser does not give a windowed plug-in a native window, because the Mac OS platform does not support child windows. Instead, the windowed plug-in draws into the graphics port associated with the the browser window, at the offset that the browser specifies.

On MS Windows and Unix, the browser creates a child window for each plug-in instance and passes it a window through **NPP_SetWindow**. On Mac OS, the application uses `NPP_SetWindow` to dedicate a rectangular part of its graphics port to each instance. On any platform, the browser should be careful not to draw in the plug-in's area, and vice versa. The data structure passed in `NPP_SetWindow` is an **NPWindow** object, which contains the coordinates of the instance's area and various platform-specific data.

Typically, the browser calls `NPP_SetWindow` after creating the instance so that the plug-in can begin drawing immediately. However, the browser can create invisible instances for which `NPP_SetWindow` is never called and a window is never created. This happens when plug-ins are invoked with an HTML `OBJECT` tag that has been hidden with special CSS rules (see **Plug-in Display Modes** in the Introduction) or with an `EMBED` tag whose the `HIDDEN` attribute has been set.

The browser should call `NPP_SetWindow` again whenever the size or position of the instance changes, passing it the same `NPWindow` object each time, but with different values.

The browser can also call `NPP_SetWindow` multiple times with different values for the window, including null. For example, if a user removes an instance from the page, The browser should call `NPP_SetWindow` with a window value of null. This value prevents the instance from drawing further until it is pasted back on the page and `NPP_SetWindow` is called again with a new value.

- **Mac OS**
- **Windows**
- **Unix**

Mac OS

On Mac OS, the browser passes an `NP_Port` structure in the window field of the `NPWindow` structure. This structure contains a pointer to the graphics port (`CGraphPtr`) into which the plug-in instance should draw and the x- and y-coordinates of the upper-left corner of this port. The plug-in can use these coordinates to call `SetOrigin(portx, porty)` to place the upper-left corner of its rectangle at (0,0). The Mac OS `GrafPort` structure's `clipRgn` field should be set to the clipping rectangle for the instance in port coordinates.

Because the plug-in and the browser share the same graphics port, they share the responsibility for managing it correctly. The browser sets up the port for the plug-in before passing the plug-in an update event in two ways:

- The browser calls `SetOrigin(npport->portx, npport->porty)`. This method makes the instance's upper-left coordinate equal to (0,0).
- The browser sets the port's clip region to the region of the plug-in currently visible (not scrolled off the page, obscured by floating palettes, or otherwise hidden).

However, for the plug-in to draw at any other time, for example, to highlight on a mouse-down event or draw animation at idle time, it must save the current setting of the port, set up its drawing environment as appropriate, draw, and then restore the port to the previous settings. In this case, the plug-in makes it unnecessary for the browser to save and restore its port settings before and after every call into the plug-in.

The browser and the plug-in can both install Drag Manager handlers for the shared port. Because the Drag Manager calls both handlers no matter where the cursor is, the browser does not show the drag highlight when the cursor is over an instance rectangle. Also, the browser does nothing when a drop occurs within an instance rectangle. The plug-in can then show the drag highlight and handle drops when they occur within the instance rectangle.

The browser is also responsible for sending the plug-in all events targeted to an instance, such as mouse clicks when the cursor is within the instance rectangle or suspend and resume events when the application is switched in and out. Events are sent to the plug-in with a call to `NPP_HandleEvent`; for a complete list of event types, see the reference entry for `NPEvent`.

Windows

On Windows, the browser registers a window class and creates an instance of that class for the plug-in instance. The plug-in can then subclass the window to receive any events it needs. If the plug-in needs to receive periodic time messages (for example, for animation), it should use a timer or a separate thread.

Unix

On Unix, the browser creates a Motif Drawing Area widget for the instance and passes the window ID of the widget in the window field of `NPWindow`. Additionally, the browser creates an `NPSetWindowCallbackStruct` object and passes it in the `ws_info` field of `NPWindow`. As on Windows, the plug-in can receive all events for the instance, in this case through the widget. If the plug-in needs to receive periodic time messages, it should install a timer or fork a thread.

Event Handling for Windowed Plug-ins

All imaging and user interface events for a windowed plug-in instance are handled according to the windowing system of its native platform. The Plug-in API provides a native window handle within which an instance does its drawing through the API call `NPP_SetWindow`. `NPP_SetWindow` passes the instance an `NPWindow` object containing the native window handle.

On Windows and Unix, each instance receives its own child window within the browser window hierarchy, and imaging and event processing are relative to this window. The Mac OS does not support child windows. The native window is shared between the instance and the browser. The instance must restrict its drawing to a specified area of the shared window, and it must always save the current settings, set up the drawing environment, and restore the shared drawing environment to the previous settings. On Mac OS, events are explicitly provided to the instance by `NPP_HandleEvent`.

Windowless Plug-ins

A windowless plug-in does not require a native window to draw into. Instead it draws into a drawable (HDC on Windows or CGrafPtr on Mac OS), which can either be on-screen or off-screen.

Windowless plug-ins provide the plug-in writer with some significant design possibilities:

- You can place a windowless plug-in within a section; other sections can exist both above and below it.
- You can create transparent plug-ins. In this case, the browser draws the part of the page that exists behind the plug-in. The windowless plug-in draws only the parts of itself that are opaque. This way, the plug-in can draw an irregularly shaped area, such as a figure, or text over the existing background.
- The browser supports off-screen drawing of plug-ins. This makes it possible to manipulate plug-in contents. For example, a 3D application could use the contents of a plug-in as a texture map.

Because windowless plug-ins can be layered or drawn to arbitrary drawables, the browser (as opposed to the native windowing system) is responsible for controlling both their drawing and their event handling.

See the following items for more information on controlling the drawing of the plug-in instance:

- **Specifying That a Plug-in Is Windowless**
- **Invalidating the Drawing Area**
- **Forcing a Paint Message**
- **Making a Plug-in Opaque**
- **Making a Plug-in Transparent**
- **Creating Pop-up Menus and Dialog Boxes**
- **Event Handling for Windowless Plug-ins**

Specifying That a Plug-in Is Windowless

To specify that a plug-in is windowless, use the **NPN_SetValue** method.

```
NPError NPN_SetValue(NPP instance,
NPPVariable variable, void *value);
```

The instance parameter represents the current plug-in. The variable parameter contains plug-in information to set. The value parameter returns the name of the plug-in.

To specify that a plug-in is windowless, use `NPN_SetValue` with `NPPVpluginWindowBool` as the value of variable and false as the value of value. The plug-in makes this call from its `NPP_New` method. If a plug-in does not make this call, it is considered a windowed plug-in.

```
NPN_SetValue
typedef enum {
...
...
NPPVpluginWindowBool,
NPPVpluginTransparentBool
} NPPVariable;
NPError    NPN_SetValue(NPP instance, NPPVariable
variable, void *value);
```

Invalidating the Drawing Area

Before it can repaint or refresh part of its drawing area, a windowless plug-in must first invalidate the area with either of these browser methods: `NPN_InvalidateRect` or `NPN_InvalidateRegion`. Both methods perform the same operations:

- They invalidate the specified drawing area prior to repainting or refreshing.
- They pass an update event or a paint message to the plug-in.

The browser redraws invalid areas of the document and windowless plug-ins at regularly timed intervals. To force a paint message, the plug-in can call `NPN_ForceRedraw` after calling one of the invalidate methods. If a plug-in calls one of these methods, it receives a paint message asynchronously.

```
void NPN_InvalidateRect(NPP instance,
  NPRect *invalidRect);
void NPN_InvalidateRegion(NPP instance,
  NPRegion invalidRegion);
```

The `instance` parameter represents the current plug-in. The `invalidRect` and `invalidRegion` parameters represent the area to invalidate, specified in a coordinate system whose origin is at the top left of the plug-in.

Both methods cause the NPP_HandleEvent method to pass an update event or a paint message to the plug-in.

```
#ifdef XP_MAC
typedef RgnHandle NPRegion;
#elif defined(XP_WIN)
typedef HRGN NPRegion;
#elif defined(XP_UNIX)
typedef Region NPRegion;
#else
typedef void* NPRegion;
#endif /* XP_MAC */

void NPN_InvalidateRect(NPP instance, NPRect
*invalidRect);
void NPN_InvalidateRegion(NPP instance, NPRegion
invalidRegion);
```

Forcing a Paint Message

Windowed and windowless plug-ins have different drawing models. A windowed plug-in determines when it draws, whereas a windowless plug-in draws in response to a paint message from the browser. A plug-in can call NPN_ForceRedraw to force a paint message synchronously, once an area has been invalidated with NPN_InvalidateRect or NPN_InvalidateRegion.

```
void NPN_ForceRedraw(NPP instance);
```

This method results in a synchronous update event or paint message for the plug-in.

A plug-in must not draw into its drawable unless it receives a paint message. It does not need to call the platform-specific function to begin painting within a window. That is, the plug-in does not call BeginPaint on Windows or BeginUpdate on Mac OS.

Windows

The plug-in receives a WM_PAINT message. The lParam parameter of WM_PAINT holds a pointer to an NPRect structure specifying the bounding box of the update area. Because the plug-in and the browser share the same HDC, the plug-in must save the current settings on the HDC, set up its own environment, draw itself, and restore the HDC to the previous settings. The HDC settings must be restored whenever control returns to the the browser, either before returning from NPP_HandleEvent or before calling a drawing- related browser-side method.

Mac OS

The plug-in receives an update event. The clip region of the drawable's `CGrafPtr` port is set to the update region. As is the case for windowed plug- ins on Mac OS, the plug-in must first save the current settings of the port, setting up the drawing environment as appropriate, drawing, and restoring the port to the previous setting. This should happen before the plug-in returns from `NP_HandleEvent` or before the plug-in calls a drawing-related browser method.

Making a Plug-in Opaque

A windowless plug-in is opaque if it has no transparent areas. When the browser generates a paint message for the plug-in, it assumes that the plug-in is responsible for painting the entire area to be updated. Because the browser does not need to draw the background behind the plug-in, opaque windowless plug-ins are considerably more efficient than transparent plug-ins.

A windowless plug-in is transparent by default. To make a transparent plug-in opaque, call `NPN_SetValue` to set `NPPVpluginTransparentBool` to `false`. The plug-in can call this method any time after specifying that it is a windowless plug-in.

Making a Plug-in Transparent

A windowless plug-in is transparent if it has transparent areas. Here are two examples of plug-ins that have transparent areas:

- plug-in that is smaller than the area specified by the enclosing `OBJECT` or `EMBED` tag
- plug-in with nonrectangular boundaries

The browser is responsible for rendering the background of a transparent windowless plug-in. Before generating a paint message for the plug-in, the browser makes sure that the background is already drawn into the area to be updated. The plug-in can then draw the part of the update region that corresponds to its opaque areas. This ensures that the transparent areas of the plug-in are always valid.

Windowless plug-ins are transparent by default. If you want to make an opaque windowless plug-in transparent, call the `NPN_SetValue` method and set `NPPVpluginTransparentBool` to the value `true`. The plug-in can call this method any time after specifying that it is a windowless plug-in.

Creating Pop-up Menus and Dialog Boxes

MS Windows only

A windowless plug-in does not draw in its own native window. Instead, it draws directly in the drawable given to it. This behavior presents a problem if you need to display pop-up menus and modal dialog boxes in a plug-in; a plug-in needs a parent window in order to create windows like these.

To deal with this problem on Windows, use `NPN_GetValue` to find out where the plug-in draws. Use `NPNVnetscapeWindow` as the value for the variable parameter.

```
NPError NPN_GetValue(NPP instance,
NPNVariable variable, void *value);
```

The instance parameter represents the current plug-in. The variable parameter contains the information the call is requesting, in this case `NPNVnetscapeWindow` (the native window in which plug-in drawing occurs). The requested information, a value of type `HWND`, is returned in the value parameter.

In many cases, a plug-in may still have to create its own window (a transparent child window of the the browser window) to act as the owner window for pop-up menus and modal dialog boxes. You can give this transparent child window its own WindowProc process. The plug-in can use this to deal with `WM_COMMAND` messages sent to it as a result of tracking the pop-up menu or modal dialog box.

Event Handling for Windowless Plug-ins

On all platforms, platform-specific events are passed to windowless plug-ins through the `NPP_HandleEvent` method. The plug-in must return true from `NPP_HandleEvent` if it has handled the event and false if it has not. Mac OS uses this mechanism for both windowed and windowless plug-ins; on this platform, `NPP_HandleEvent` is the only way the plug-in can receive events from its host application.

```
int16 NPP_HandleEvent(NPP instance, NPEvent *event);
```

The `instance` parameter represents the current plug-in. For a list of event types the application is responsible for delivering to the plug-in, see the **NPEvent** structure.

This code shows the specific data passed through this method for each platform:

```
#ifdef XP_MAC
typedef EventRecord NPEvent;
#elif defined(XP_WIN)
typedef struct _NPEvent {
int16    event;
int16    wParam;
int32    lParam;
} NPEvent;
#elif defined(XP_UNIX)
typedef XEvent NPEvent;
#else
typedef void NPEvent;
#endif /* XP_MAC */

int16 NPP_HandleEvent(NPP instance, NPEvent* event);
```

On Mac OS, when `NPP_HandleEvent` is called, the current port is set up correctly so that its origin matches the upper-left corner of the plug-in. A plug-in does not need to set up the current port for mouse coordinate translation.

Streams

This chapter describes using Plug-in API functions to receive and send streams.

Streams are objects that represent URLs and the data they contain, or data sent by a plug-in without an associated URL. Although a single stream is associated with one specific instance of a plug-in, a plug-in can have more than one stream object per instance. Streams can be produced by the browser and consumed by a plug-in instance, or produced by an instance and consumed by the browser. Each stream has an associated MIME type identifying the format of the data in the stream.

Streams produced by the browser can be automatically sent to or requested by the plug-in instance. The browser calls the Plug-in methods **NPP_NewStream**, **NPP_WriteReady**, **NPP_Write**, and **NPP_DestroyStream** to, respectively, create a stream, find out how much data the plug-in can handle, push data into the stream, and delete it.

The plug-in instance selects a transmission mode for streams produced by the browser. Stream data can be pushed by the browser, pulled by the plug-in, or saved to a local file and passed to the plug-in.

- Normal mode: The browser uses the `NPP_Write` method to "push" stream data to the instance incrementally as it is available.
- Random-access mode: The plug-in calls the `NPN_RequestRead` method to "pull" stream data. In general, this mode is more expensive, because the entire stream must be downloaded to a temporary file before use unless the stream comes from a local file or an HTTP server that supports the proposed byte-range extension to HTTP.
- File mode: The browser saves the entire stream to a local file and passes the file path to the plug-in instance through the `NPP_StreamAsFile` method. Use this feature only as a last resort; plug-ins should implement an incremental stream-based interface wherever possible.

Streams sent by the plug-in to the browser are like normal-mode streams produced by the browser, but in reverse. In normal-mode streams, the browser calls the plug-in to tell it when a stream is created and to push more data. In contrast, for streams

produced by the plug-in, the plug-in calls the Plug-in API methods `NPP_NewStream`, `NPP_Write`, and `NPP_DestroyStream` to create a stream, push data into it, and delete it.

- **Receiving a Stream**
- **Sending a Stream**

Receiving a Stream

When the browser sends a data stream to the plug-in, it has several tasks to perform:

- **Telling the Plug-in When a Stream Is Created**
- **Telling the Plug-in When a Stream Is Deleted**
- **Finding Out How Much Data the Plug-in Can Accept**
- **Writing the Stream to the Plug-in**
- **Sending the Stream in Random-Access Mode**
- **Sending the Stream in File Mode**

Telling the Plug-in When a Stream Is Created

To tell the plug-in instance when a new stream is created, the browser calls the `NPP_NewStream` method. This method also determines which mode it should use to send data to the plug-in. The browser can create a stream for several different types of data:

- for the file specified in the `SRC` attribute of the `EMBED` tag
- for a data file
- for a full-page instance

The `NPP_NewStream` method has the following syntax:

```
NPError NPP_NewStream(NPP instance, NPMIMEType type,
NPStream *stream, NPBool seekable, uint16* stype);
```

The `instance` parameter refers to the plug-in instance receiving the stream; the `type` parameter represents the stream's MIME type.

The `stream` parameter is a pointer to the new stream, which is valid until the stream is destroyed.

The `seekable` parameter specifies whether the stream is seekable (`true`) or not (`false`). Seekable streams support random access (for example, local files or HTTP servers that support byte-range requests).

The plug-in can set the `output` parameter `type` to one of these transmission modes:

- `NP_NORMAL` (Default): The plug-in can process the data progressively as it arrives from the network or file system through series of calls to `NPP_WriteReady` and `NPP_Write`.
- `NP_ASFILEONLY`: This plug-in gets full random access to the data using platform-specific file operations. The browser saves stream data to a local file, and, when the stream is complete, delivers the path of the file through a call to `NPP_StreamAsFile`.
- `NP_ASFILE`: This mode is like `NP_ASFILEONLY` except that data is delivered to the plug-in as it is saved to the file, through a series of calls to `NPP_Write`. You should use `NP_ASFILEONLY` whenever possible in preference to `NP_ASFILE`, which is less efficient because it uses successive calls to `NPP_Write` to send the data.
- `NP_SEEK`: The plug-in instance can randomly access stream data as needed, through calls to `NPN_RequestRead`. If the stream is not seekable, these requests are fulfilled only when all the data has been read and stored in the cache.

Once all data in the stream has been written to the plug-in, the stream is destroyed. To do this, either the browser can call `NPP_DestroyStream` or the plug-in can call `NPN_DestroyStream`. This applies to all plug-in modes except `NP_SEEK`.

NOTE: A plug-in can also use the NPN_GetURL method to request a stream for an arbitrary URL.

Telling the Plug-in When a Stream Is Deleted

The browser calls the `NPP_DestroyStream` method when it completes the stream sent to the plug-in, either successfully or abnormally. Once the plug-in returns from this method, the browser deletes the `NPStream` object. The plug-in can terminate the stream itself by calling `NPN_DestroyStream`.

You should delete any private data allocated in the plug-in's stream->pdata field when you destroy a stream. The plug-in can store private data associated with the stream in stream->pdata. The browser stores private data in stream->ndata; this value should not be changed by the plug-in.

```
NPError NPP_DestroyStream(NPP instance,
NPStream *stream, NPError reason);
```

The `instance` parameter is the current plug-in instance; the `stream` parameter specifies the stream to be deleted.

The `reason` parameter specifies why the stream was destroyed. It can have one of these values:

- `NPRES_DONE` (Most common): Normal completion; all data was sent to the instance.
- `NPRES_USER_BREAK`: The user canceled the stream directly by clicking the Stop button or indirectly by some action, such as by deleting the instance or initiating higher-priority network operations.
- `NPRES_NETWORK_ERR`: The stream failed because of problems with the network, disk I/O error, lack of memory, or some other problem.

Finding Out How Much Data the Plug-in Can Accept

After a call to `NPP_NewStream` and before writing data to the plug-in, the browser calls `NPP_WriteReady` to determine the maximum number of bytes that the plug-in can consume. This function allows the browser to send only as much data to the instance as it can handle at one time, and it helps both the browser and the plug-in to use their resources efficiently.

After a call to `NPP_NewStream`, in which the plug-in requested a normal-mode stream, the browser delivers the data in the stream progressively in a series of calls to `NPP_WriteReady` and `NPP_Write`. The browser calls `NPP_WriteReady` before each call to `NPP_Write`.

The value returned by `NPP_WriteReady` indicates how many bytes the plug-in instance can accept for this stream. If the plug-in allocates memory for the entire stream at once, it can return a large number. This number tells the browser that it can pass as much data to the instance as possible in a single call to `NPP_Write`. The browser can write a smaller amount of data if desired or necessary (for example, if only 8K of data is available in a network buffer).

For instance, suppose the plug-in allocates, in NPP_NewStream, an 8K buffer to hold the data written from that stream. In the first call, NPP_WriteReady could return 8192, resulting in a call to NPP_Write with a buffer of up to 8K bytes. After this data is copied from the browser's buffer to the plug-in's buffer, the plug-in begins to process the data asynchronously. At the next NPP_WriteReady call, only half of the data has been processed. To avoid allocating additional buffers, the plug-in could return 4096, resulting in a call to NPP_Write with a buffer of up to 4K bytes.

The buffer passed to NPP_Write may accommodate more bytes than the maximum number returned from NPP_WriteReady . This maximum is only a promise to consume a certain amount of data from the buffer, not an upper limit on the buffer size. In the example above, suppose that the plug-in allocates an 8K buffer and returns 8192 from NPP_WriteReady. If the plug-in gets 10000 bytes from the browser in a subsequent call to NPP_Write, the plug-in should copy the first 8192 bytes from the browser's buffer into its own buffer and return 8192 (the number of bytes actually consumed) from NPP_Write.

```
int32 NPP_WriteReady(NPP instance, NPStream *stream);
```

The instance parameter is the current plug-in instance; the stream parameter specifies the current stream.

Writing the Stream to the Plug-in

The next step is to write the data to a plug-in from a stream. After a call to NPP_NewStream, in which the plug-in requested a normal-mode stream, the browser delivers the data in the stream progressively in a series of calls to NPP_WriteReady and NPP_Write.

The NPP_Write function should return the number of bytes consumed by the instance. If this is a negative number, the browser calls NPP_DestroyStream to destroy the stream. If the number returned is smaller than the size of the buffer, the browser sends the remaining data in the buffer to the plug-in through repeated calls to NPP_WriteReady and NPP_Write.

```
int32 NPP_Write(NPP instance, NPStream *stream,
int32 offset, int32 len, void *buf);
```

The instance parameter is the current plug-in instance; the stream parameter specifies the current stream. The offset parameter specifies the offset, in bytes, of buf from the beginning of the data in the stream. The len parameter specifies the

length, in bytes, of buf , the buffer of data (delivered by the stream). The buffer allocated by the browser is deleted after returning from the function, so the plug-in must make a copy of the data it needs to keep.

As an example, suppose that a plug-in (and the HTTP server) supports byte-range requests, and that the browser is in the process of pushing data to the plug-in. If the user now requests a specific page of the document, the plug-in calls `NPN_RequestRead` with a list of byte ranges. The open stream is converted from normal mode to seek mode in an effort to pass the plug-in data that was already on the way, rather than just discarding it. All `NPP_Write` calls for streaming data eventually stop, and `NPP_Write` calls will be completed only for data requested with `NPN_RequestRead`.

The browser does not create a new stream for each byte range it requests. Instead, additional `NPP_WriteReady` and `NPP_Write` calls occur on the same stream. An individual call to `NPN_RequestRead` can request discontiguous ranges, and you can have many outstanding `NPN_RequestRead` calls. There is no guarantee that `NPP_Write` will receive requests for ranges in the same order as you requested (although this typically is the case; the server controls the order). So, you'll need to pay attention to the offsets as data is being written.

The stream processes all byte-range requests, and then is placed in seek mode (either explicitly in `NPP_NewStream`, or implicitly by a call to `NPN_RequestRead`). It remains open until the plug-in closes it by calling `NPN_DestroyStream`, or until the instance is destroyed.

NOTE: If you want to be sure that the `NPN_*Stream` functions are called in the order you want and behave the way you expect, combine `NPN_NewStream`, `NPN_Write`, and `NPN_Destroy_Stream` in the same callback.

Sending the Stream in Random-Access Mode

In random-access mode, the plug-in "pulls" stream data by calling the `NPN_RequestRead` method. The browser must download the entire stream to a temporary file before it can be used, unless the stream comes from a local file or an HTTP server that supports the proposed byte-range extension to HTTP. This mode consumes more resources than the others.

Random-access mode is determined in `NPP_NewStream` by setting the mode `NP_SEEK`. This mode gives the plug-in instance random access to stream data as needed, through calls to `NPN_RequestRead`. If the stream is not seekable, these requests are fulfilled only when all the data has been read and stored in the cache.

The NPN_RequestRead method requests a range of bytes from a seekable stream. Typically, the only streams that are seekable are from data that is in memory or on the disk, or from HTTP servers that support byte-range requests.

- For streams that are not in NP_SEEK mode: The plug-in can call NPN_RequestRead as long as the stream is inherently seekable; NPN_RequestRead automatically changes the mode to NP_SEEK.
- For streams that are not inherently seekable: The stream must be put in NP_SEEK mode initially, because the browser must cache all the stream data on disk in order to access it randomly.
- For streams that are not inherently seekable and not initially in mode NP_SEEK: NPN_RequestRead returns the error code NPERR_STREAM_NOT_SEEKABLE.

The NPN_RequestRead method has the following syntax:

```
NPError NPN_RequestRead(NPStream *stream, NPByteRange
*rangeList);
```

The stream parameter is the stream from which to read bytes; the rangeList parameter specifies the range of bytes in the form of a linked list of NPByteRange objects, which the plug-in must allocate. Because these objects are copied by the browser, and so the plug-in can delete them as soon as the call to NPN_RequestRead returns.

The plug-in can request multiple ranges, either through a list of NPByteRange objects in a single call to NPN_RequestRead or through multiple calls to NPN_RequestRead. In this case, the browser can write individual ranges in any order, with any number of NPP_WriteReady and NPP_Write calls.

Sending the Stream in File Mode

If the stream is sent in file mode, the browser saves the entire stream to a local file and passes the full file path to the plug-in instance through the NPP_StreamAsFile method. Use this feature only as a last resort; plug-ins should implement an incremental stream-based interface whenever possible.

File mode is determined in NPP_NewStream by setting the mode NP_ASFILEONLY . This mode gives the plug-in full random access to the data with platform-specific file operations. The browser saves stream data to a local file, and, when the stream is complete, delivers the path of the file through a call to NPP_StreamAsFile.

NOTE: Most plug-ins that need the stream saved to a file should use NP_ASFILEONLY mode rather than the older NP_ASFILE; this mode is less efficient because it uses successive calls to NPP_Write. NPP_StreamAsFile provides the plug-in with a full path to a local file for the stream. It is a good idea to check that the file exists in the directory at the start of this method. If an error occurs during data retrieval or writing to the file, the browser passes null for the filename. If the file is created from a stream from the network, the file is locked in the the browser disk cache until the stream or its instance is destroyed.

```
void NPP_StreamAsFile(NPP instance, NPStream *stream,
const char* fname);
```

The instance parameter is the current plug-in; the stream parameter specifies the current stream. The fname parameter specifies the full path to a local file (or null if an error occurs during data retrieval or writing to the file).

Sending a Stream

When a plug-in sends a data stream to the browser, it performs several tasks. The plug-in calls the methods NPN_NewStream, NPN_Write, and NPN_DestroyStream to create a stream, push data into it, and delete it. Streams produced by a plug-in have a specific MIME type and can be sent to a particular browser window or frame for display.

- **Creating a Stream**
- **Pushing Data into the Stream**
- **Deleting the Stream**

For an example that demonstrates these processes, see "**Example of Sending a Stream**."

Creating a Stream

The plug-in calls NPN_NewStream to send a new data stream to the browser. The browser creates a new NPStream object and returns it to the plug-in as an output parameter.

The plug-in can use this stream object in subsequent NPN_Write calls to the browser. When all the plug-in data is written into the stream, the plug-in must terminate the stream and deallocate the NPStream object by calling the NPN_DestroyStream function.

```
NPError NPN_NewStream(NPP instance,
                      NPMIMEType type,
                      const char* target,
                      NPStream** stream);
```

The instance parameter is the plug-in instance that is creating the stream; the type specifies the MIME type of the stream.

The target parameter specifies the window or frame. For the possible values of named targets, see the reference entry for NPN_NewStream. The target should not be the same window.

The stream parameter represents the stream that the browser creates.

For an example that demonstrates using this function with NPN_Write and NPN_DestroyStream, see "**Example of Sending a Stream**."

Pushing Data into the Stream

After creating a stream with NPN_NewStream, the plug-in can call NPN_Write to deliver a buffer of data from the plug-in to the browser. This function returns the number of bytes written or a negative integer in case of an error during processing. NPN_Write should send as much data as is available. Unlike NPP_Write, NPN_Write has no corresponding NPN_WriteReady function.

```
int32 NPN_Write(NPP instance, NPStream *stream,
int32 len, void *buf);
```

The plug-in should terminate the stream by calling NPN_DestroyStream, when all data has been written to the stream, or in the event of an error.

The instance parameter is the current plug-in; the stream parameter is a pointer to the stream being written to. The len parameter specifies the length, in bytes, of data written to the stream. The buf parameter is a pointer to the buffer holding the data to write to the stream.

For an example that demonstrates using this function with NPN_NewStream and NPN_DestroyStream, see "**Example of Sending a Stream**."

Deleting the Stream

When the stream is complete, the plug-in calls `NPN_DestroyStream` to close and delete it. This applies to streams the plug-in creates with `NPN_NewStream` or streams created by the browser with `NPP_NewStream`.

```
NPError NPN_DestroyStream(NPP instance, NPStream* stream,
NPError reason);
```

The `instance` parameter is the current plug-in; the stream parameter specifies the stream, created by either the browser or the plug-in. The reason parameter represents the reason the stream was stopped, as follows:

- `NPRES_DONE` (most common): The stream completed normally; the plug-in sent all data to the browser.
- `NPRES_USER_BREAK`: The plug-in terminated the stream because of a user request.
- `NPRES_NETWORK_ERR`: The stream failed because of network problems.

For the complete list of codes, see "**Result Codes**."

For an example that demonstrates using this function with `NPN_NewStream` and `NPN_Write`, see "**Example of Sending a Stream**."

Example of Sending a Stream

The following code creates a new stream of HTML text displayed by the browser in a new window, writes it, and destroys the stream. Error handling has been omitted for simplicity.

```
NPStream* stream;
char* myData = "<HTML><B>This is a message from my plug-in!</B></HTML>";
int32 myLength = strlen(myData) + 1;

/* Create the stream. */
err = NPN_NewStream(instance, "text/html", "_blank", &stream);

/* Push data into the stream. */
err = NPN_Write(instance, stream, myLength, myData);

/* Delete the stream. */
err = NPN_DestroyStream(instance, stream, NPRES_DONE);
```

Your plug-in can create another instance of itself by specifying its own MIME type and a new target name in a call to `NPN_NewStream`.

URLs

This chapter describes retrieving URLs and displaying them on specified target pages, posting data to an HTTP server, uploading files to an FTP server, and sending mail.

Uniform resource locator (URL) protocols provide a means for locating and accessing resources that are available on the Internet and on intranets. Plug-ins can request and receive the data associated with URLs of any type that the browser can handle, including HTTP, FTP, news, mailto, and gopher.

The table below summarizes URLs supported by the Netscape browser. In addition, Netscape may support URLs not listed on this table.

URL Scheme	Description
`about`	Locates browser information or "fun" pages. Netscape proprietary.
`file`	(Host-specific filenames) Locates files on a specific host computer rather than an Internet resource.
`ftp`	(File Transfer Protocol) Locates files and directories on Internet hosts for file download.
`gopher`	(Gopher protocol) Locates specified items on a Gopher server.
`http`	(Hypertext Transfer Protocol) Locates resources on the Internet.
`javascript`	Executes JavaScript code that follows the URL. Netscape-specific.
`mailto`	(Electronic mail address) Locates the Internet mailing address of an individual or service.
`nethelp`	Displays a NetHelp topic in a NetHelp window. Browser-specific.
`news`	(USENET news) Locates USENET news groups or individual USENET articles.
`nntp`	(USENET news using nntp access) Locates USENET news groups or individual USENET articles; alternate to news.

prospero	(Prospero Directory Service) Locates a resource on a Prospero directory server.
telnet	(Reference to interactive sessions) Locates an interactive service.
wais	(Wide Area Information Servers) Locates WAIS databases and their documents.
wysiwyg	Placed before another URL; displays a page that JavaScript has updated using document.write.

For more information, see RFC 1738, "Uniform Resource Locators (URL).

- **Getting URLs**
- **Posting URLs**

Getting URLs

To retrieve a URL and display it on a specified target page, use the **NPN_GetURL**, **NPN_GetURLNotify**, and **NPP_URLNotify** functions. This section describes the methods and procedure used for getting the URL and displaying the page.

The plug-in uses the `NPN_GetURL` function to ask the browser to display data retrieved from a URL in a specified target window or frame, or deliver it to the plug-in instance in a new stream. This is the way that plug-ins provide hyperlinks to other documents or retrieve data from the network.

If the browser cannot locate the URL and retrieve the data, it does not create a stream for the instance; in this case, the plug-in receives notification of the result. To request a stream and receive notification of the result in all cases, use `NPN_GetURLNotify`.

For HTTP URLs, the browser resolves `NPN_GetURL` as the HTTP server method GET, which requests URL objects.

Note that `NPN_GetURL` is typically asynchronous: it returns immediately and only later handles the request, such as displaying the URL or creating the stream for the instance and writing the data. For this reason as well, calling `NPN_GetURLNotify` may be more useful than `NPN_GetURL`; the plug-in is notified upon either successful or unsuccessful completion of the request.

```
NPError NPN_GetURL(NPP instance, const char *url, const char *target);
```

The instance parameter represents the current plug-in instance. The url parameter is the URL of the request, which can be of any type, including HTTP, FTP, news, mailto, or gopher.

The target parameter represents the destination where the URL will be displayed, a window or frame. If target refers to the window or frame containing the plug-in instance, it is destroyed and the plug-in may be unloaded. If the target parameter is set to null, the application creates a new stream and delivers the data to the plug-in instance, through calls to **NPP_NewStream**, **NPP_WriteReady** and **NPP_Write**, and **NPP_DestroyStream**.

In general, if a URL works in the location box of the Navigator, it works as a target for `NPN_GetURL`, except for the `_self` target.

Make sure that the target matches the URL type sent to it. For example, a null target does not make sense for some URL types (such as mailto). For some recommendations to help you with target parameter choice, see the reference entry for **NPN_GetURL**.

The `NPN_GetURLNotify` method acts like `NPN_GetURL`. Both request the creation of a new stream with the contents of the specified URL, and, in addition, `NPN_GetURLNotify` notifies the plug-in of the successful or unsuccessful completion of the request. The browser notifies the plug-in by calling the plug-in's `NPP_URLNotify` function and passing it the `notifyData` value, which may be used to track multiple requests.

`NPN_GetURLNotify` handles the URL request asynchronously. It returns immediately and only later handles the request and calls `NPP_URLNotify`. Without this notification, the plug-in cannot tell whether a request with a null target failed or a request with a non-null target was completed.

```
NPError NPN_GetURLNotify(NPP instance, const char* url,
                         const char* target, void*
notifyData);
```

The instance, url, and target parameters have the same definitions as those of `NPN_GetURL`. The notifyData parameter contains private plug-in data that can be used to associate the request with the subsequent `NPP_URLNotify` call (which returns this value) and/or to pass a pointer to some request-related payload.

If a request is not completed successfully (for example, because the URL is invalid or a HTTP server is down), the browser should call `NPP_URLNotify` as soon as possible. If a request completes successfully, and the target is non-null, the browser calls `NPP_URLNotify` after it has finished loading the URL. If the target is null, it calls `NPP_URLNotify` after calling `NPP_DestroyStream` to close the stream.

Both the `NPN_GetURLNotify` and `NPN_PostURLNotify` functions call the `NPP_URLNotify` method to notify the plug-in of the result of a request. Both functions pass the notifyData value to `NPP_URLNotify`, which tells the plug-in that the URL request was completed and the reason for completion.

```
void NPP_URLNotify(NPP instance, const char* url,
                   NPReason reason, void* notifyData);
```

The instance and url parameters have the same definitions as those of `NPN_GetURL`. The `notifyData` parameter contains the private plug-in data passed to the corresponding call to `NPN_GetURLNotify` and `NPN_PostURLNotify`.

Getting the URL and Displaying the Page

To retrieve a URL and display it on a specified target page, you use the `NPN_GetURL` and `NPN_GetURLNotify` functions. The URL can be displayed in the same window or frame, a new window, or a different window or frame, depending on the value of the target parameter. Specify the display target with one of these special target names:

- `_blank` or `_new`: Load the URL in a new blank unnamed window. Safest target, even though, when used with a mailto or news URL, this creates an extra blank the browser instance.
- `_self` or `_current`: Load the URL into the same window the plug-in instance occupies. If this target refers to the window or frame containing the instance, the instance is destroyed and the plug-in may be unloaded.
- `_parent`: Load the URL into the immediate `FRAMESET` parent of the plug-in instance document. If the plug-in instance document has no parent, the default is _self.
- `_top`: Load the URL into the plug-in instance window. The default is `_self`, if the plug-in instance document is already at the top. Use for breaking out of a deep frame nesting.

Be careful when you assign a target. If the target refers to the window or frame containing the instance or one of its parents/ancestors, the instance is destroyed and the plug-in may be unloaded.

Here's an example of getting a URL: A plug-in instance draws a button that acts like a link to another web page. When the user clicks the button, the plug-in calls `NPN_GetURL` to go to the page.

```
err = NPN_GetURL(
  instance, "http://home.netscape.com/", "_blank");
```

Posting URLs

- **Posting Data to an HTTP Server**
- **Uploading Files to an FTP Server**
- **Sending Mail**

The plug-in calls **NPN_PostURL** to post data from a file or buffer to a URL. This function is the counterpart of **NPN_GetURL**.

- `NPN_PostURL` writes data from a file or buffer to the URL and either displays the server response in the target window or delivers it to the plug-in.
- `NPN_GetURL` reads data from the URL and either displays it in the target window or delivers it to the plug-in.

For HTTP URLs only, the browser resolves this method as the HTTP server method `POST`, which transmits data to the server.

You can use `NPN_PostURL` to post data to a URL from a memory buffer or file. The result from the server can also be sent to a particular the browser window or frame for display, or delivered to the plug-in instance in a new stream. Plug-ins can use this capability to post form data to CGI scripts using HTTP or upload files to a remote server using FTP.

The browser resolves this method as the HTTP server method POST , which transmits data to the server. The data to post can be contained either in a local temporary file or a new memory buffer. To post a file, set the flag file to true, the buffer buf to the path name string for a file, and len to the length of the path string. The file-type URL prefix "file://" is optional.

NPN_PostURL is typically asynchronous: it returns immediately and only later handles the request and calls NPP_Notify (which, in turn, calls NPP_URLNotify).

```
NPError NPN_PostURL(NPP instance, const char *url,
const char *target, uint32 len,
const char *buf, NPBool file);
```

The instance, url, and target parameters have the same definitions as those of NPN_GetURL.

The buf parameter identifies a local temporary file or data buffer that contains the data to post.

Windows and Mac

If a file is posted with any protocol other than FTP, the file must be text with Unix-style line breaks ('\n' separators only).

NPN_PostURL works identically with buffers and files. To post data from a memory buffer, set the flag file to false, the buffer buf to the data to post, and len to the length of the buffer.

Possible URL types include http (similar to an HTML form submission), mailto (sending mail), news (posting a news article), and ftp (uploading a file). For protocols in which the headers must be distinguished from the body, such as http , the buffer or file should contain the headers, followed by a blank line, then the body. If no custom headers are required, simply add a blank line ('\n') to the beginning of the file or buffer.

NOTE: You cannot use NPN_PostURL to specify headers (even a blank line) in a memory buffer. To do this, use NPN_PostURLNotify for this purpose. § The NPN_PostURLNotify function has all the same capabilities and works like NPN_PostURL in most ways except that (1) it supports specifying headers when posting a memory buffer, and (2) it calls NPP_URLNotify upon successful or unsuccessful completion of the request. NPN_PostURLNotify is typically asynchronous: it returns immediately and only later handles the request and calls NPP_URLNotify.

```
NPError NPN_PostURLNotify(
  NPP instance, const char *url,
  const char *target, uint32 len,
  const char *buf, NPBool file, void* notifyData
);
```

The parameters of this function have the same definitions as those of `NPN_PostURL`. The notifyData parameter contains plug-in-private data passed by `NPP_URLNotify` and may be used for tracking multiple posts.

Posting Data to an HTTP Server

The following code posts two name-value pairs to a CGI script through HTTP. The response from the server is displayed in a new window.

```
char* myData = "Content Type:\tapplication/
               x-www-form-urlencoded\nContent
Length:\t25\n\nname1=value1&name2=value2\n";
uint32 myLength = strlen(myData) + 1;

err = NPN_PostURL(instance, "http://
hoohoo.ncsa.uiuc.edu/
               cgi-bin/post-query","_blank", myLength,
myData, FALSE);
```

Uploading Files to an FTP Server

Plug-ins can use `NPN_PostURL` or `NPN_PostURLNotify` to upload files to a remote server using FTP. This example uploads a file from the root of the local file system to an FTP server and displays the response in a frame named response:

```
char* myData = "file:///c\/myDirectory/myFileName";
uint32 myLength = strlen(myData) + 1;

err = NPN_PostURL(instance, "ftp://
fred@ftp.somewhere.com/pub/",
                  "response", myLength, myData, TRUE);
```

Sending Mail

A plug-in can send an email message using `NPN_PostURL` or `NPN_PostURLNotify`. The following code sends a mail message with the default headers from the client machine.

```
char* myData = "\nHi Fred, this is a message from my
plug-in!";
uint32 myLength = strlen(myData) + 1;

err = NPN_PostURLNotify(instance,
"mailto:fred@somewhere.com",
NULL, myLength, myData, FALSE);
```

The example starts by defining the mail message, `myData`, and its length, `myLength`. It sends myData and myLength to the mailto URL mailto:fred@somewhere.com. The target window for displaying the message is null in the example. Normally, using a null target window causes the response to be delivered from the server to the plug-in instance in a new stream, but no response is expected for a mailto URL.

You cannot use either of these functions to set the body or attachments of an email message.

Memory

This chapter describes the Plug-in API functions that allocate and free memory as needed by the plug-in.

Because plug-ins share memory space with the browser, they can take advantage of any customized memory-allocation scheme the browser has. Browser memory schemes may be more efficient than standard OS memory functions, and can give the browser flexibility in the way it manages memory. In addition, the plug-in usually has the option of using its own memory functions.

The methods that handle memory belong to the browser group of methods.

- **NPN_MemAlloc** allocates memory from the browser's memory space. Use this function to allocate memory dynamically.
- **NPN_MemFree** requests that the browser free a specified block of memory. Use this function to free memory allocated with `NPN_MemAlloc`.
- **NPN_MemFlush** requests the browser to free up a specified amount of memory if not enough is currently available for the plug-in's requirements.

Allocating and Freeing Memory

To allocate memory and free memory, use these paired functions:

- `NPN_MemAlloc` allocates a specified amount of memory in the browser's memory space.
- `NPN_MemFree` deallocates a block of memory allocated using `NPN_MemAlloc`.

The plug-in can call the Plug-in API `NPN_MemAlloc` function instead of the standard malloc function to allocate dynamic memory. Using `NPN_MemAlloc` offers several advantages to the plug-in.

- A call to `NPN_MemAlloc` is more likely to succeed. The browser may be able to deallocate nonessential memory structures in response to a request.

- NPN_MemAlloc uses the browser's customized memory-allocation scheme, which is typically faster and causes less fragmentation than the standard OS memory functions.
- If the plug-in uses NPN_MemAlloc, the browser is able to manage memory more efficiently because it knows how much memory the plug-in is using at any given time.

Mac OS

The Mac OS browser frequently fills its memory partition with cached data that is purged only as necessary. Since NPN_MemAlloc automatically frees cached information if necessary to fulfill a request for memory, calls to NPN_MemAlloc may succeed where direct calls to NewPtr fail.

The NPN_MemAlloc method has the following syntax:

```
void *NPN_MemAlloc (uint32 size);
```

The size parameter is an unsigned long integer that represents the amount of memory, in bytes, to allocate in the browser's memory space. This function returns a pointer to the allocated memory or null if not enough memory is available.

The NPN_MemFree method deallocates a block of memory that was allocated using NPN_MemAlloc only. NPN_MemFree does not free memory allocated by other means.

```
void NPN_MemFree (void *ptr);
```

The ptr parameter represents a block of memory previously allocated using NPN_MemAlloc.

Flushing Memory (Mac OS only)

The NPN_MemFlush method frees a specified amount of memory. Normally, plug-ins should use NPN_MemAlloc , which automatically frees nonessential memory if necessary to fulfill the request. For Communicator 4.0 and later versions, this function is not necessary for the Mac OS platform; NPN_MemAlloc now performs memory flushing internally. You need to use NPN_MemFlush only when it is not possible to call NPN_MemAlloc, for example, when calling system methods that allocate memory indirectly. If NPN_MemAlloc is called, calls to NPN_MemFlush have no effect.

For example, suppose that the plug-in calls `NewGWorld`, and that the call fails because of insufficient memory. The plug-in should try calling `NPN_MemFlush` to free enough memory. If `NPN_MemFlush` returns a value indicating that enough memory was freed, the plug-in can call NewGWorld again. Calling `NPN_MemFlush` is particularly important to systems with small amounts of RAM and with virtual memory turned off.

To request that the browser free as much memory as possible, call `NPN_MemFlush` repeatedly until it returns 0.

```
uint32 NPN_MemFlush(uint32 size);
```

The size parameter is an unsigned long integer that represents the amount of memory, in bytes, to free in the browser's memory space. This function returns the amount of freed memory, in bytes, or 0 if no memory could be freed.

Version, UI, and Status Information

This chapter describes the functions that allow a plug-in to display a message on the status line, get agent information, and check on the current version of the Plug-in API and the browser.

- **Displaying a Status Line Message**
- **Getting Agent Information**
- **Getting the Current Version**
- **Finding Out if a Feature Exists**
- **Reloading a Plug-in**

Displaying a Status Line Message

Users are accustomed to checking the UI status line at the bottom of the browser window for updates on the progress of an operation or the URL of a link on the page. You can also use the status line to notify the user of plug-in-related information. The user might appreciate seeing the percentage completed of the current operation or the URL of a button or other link object when the cursor is over it, all of which the browser shows. In fact, your plug-in interface should be consistent with the rest of the browser in this way.

To accomplish this, the plug-in calls the **NPN_Status** method to display your message on the status line.

```
void NPN_Status(NPP instance, const char *message);
```

The `instance` parameter is the current plug-in instance, that is, the one that the status message belongs to. In the `message` parameter, pass the string you want to display on the status line.

The browser always displays the last status line message it receives, regardless of the message source. For this reason, your message is always displayed, but you have no control over how long it stays in the status line before another message replaces it. You should use a different method to display messages that the user needs to see, such as error messages.

Getting Agent Information

A plug-in can check which browser is running on the user's current system. Browsers communicate with HTTP servers, which store agent software name, version, and operating system in a user_agent field. If you want to gather usage statistics or just find out the version of your plug-in's host browser, this information can help you.

The plug-in calls the **NPN_UserAgent** method to retrieve the contents of the `user_agent` field.

```
const char* NPN_UserAgent(NPP instance);
```

The `instance` parameter represents the current plug-in instance. This function returns a string that contains the `user_agent` field of the browser.

Getting the Current Version

Your plug-in should make sure, possibly during initialization, that the version of the Plug-in API it is using is compatible with the version the browser is using. To do so, it must find the major and minor version numbers, which are determined when the plug-in and Navigator are compiled, and compare them. If the versions are not compatible, the plug-in can let the user know. The plug-in can also use the version number to find out whether a particular feature exists on the version of the browser that the plug-in is running in.

The browser and Plug-in API major version numbers represent code release numbers, and their minor version numbers represent point release numbers. For example, Plug-in API version 6.03 has a major version number of 6 and a point release number of 3.

Differing version numbers may mean that the current Plug-in API and the browser versions are incompatible. Changes to the minor version numbers indicate a smaller difference than changes to the major version. Changes to the major version numbers probably indicate incompatibility.

The plug-in calls the **NPN_Version** method to check for changes in major and minor Plug-in API version numbers. It gets the values from the plug-in rather than from the browser.

```
void NPN_Version(int *plugin_major,
int *plugin_minor,
int *netscape_major,
int *netscape_minor);
```

This function returns the plug-in version number in plugin_major, the plug-in point release number in `plugin_minor`, the browser version number in `netscape_major`, and the browser point release number in `netscape_minor`.

This code declares variables to hold the version numbers and calls `NPN_Version` to return the major and minor version numbers for the browser and the Plug-in API.

```
int plugin_major, plugin_minor, netscape_major,
netscape_minor; // declare variables to hold version numbers

void NPN_Version(
  &plugin_major, &plugin_minor, &netscape_major,
  &netscape_minor
); // find version numbers
```

Finding Out if a Feature Exists

A plug-in can figure out whether it is running in a version of the browser that supports a particular feature by using version or `NPVERS` constants (see **Version Feature Constants**). Each `NPVERS` constant represents a feature. The plug-in can compare the `NPVERS` constant to the version number. If the version supports the feature, the plug-in can operate according to plan. If not, the plug-in cannot use some functionality. If an essential feature is unavailable, the developer must arrange for alternative behavior, shut down the plug-in, or give the user a chance to decide what to do.

In this example, the `has_windowless` method finds out whether the current version supports windowless plug-ins. It starts by using `NPN_Version` to get the version numbers. It then uses the netscape_minor version number to find out if the windowless feature, represented by the `NPVERS_HAS_WINDOWLESS` constant, is

supported. If the method returns true, a windowless plug-in can confidently proceed. If false is returned, windowless plug-ins will not work, and the developer must provide alternatives.

```
Bool has_windowless()
{
int plugin_major, plugin_minor;
int netscape_major, netscape_minor;

/* Find the version numbers. */
NPN_Version(&plugin_major, &plugin_minor,
&netscape_major, &netscape_minor);

/* Use the netscape_minor version number: */
/* Does this version support the windowless feature? */
if (netscape_minor < NPVERS_HAS_WINDOWLESS) {
/* Plug-in is running in a version of the Navigator */
/* that does not support windowless plug-ins. */
return FALSE;
}

else
/* Plug-in is running in a Navigator version */
/* that has windowless support */
return TRUE;
}
```

Reloading a Plug-in

When the browser starts up, it loads all the plug-ins it finds in the Plugins directory for the platform. If you call **NPN_ReloadPlugins**, the browser reloads all plug-ins in the Plugins directory without restarting. This causes the browser to install a new plug-in and load it, or remove a plug-in, without having to restart. Consider using this function as part of the plug-in's SmartUpdate process.

```
void NPN_ReloadPlugins(NPBool reloadPages);
```

The `reloadPages` parameter is a boolean that indicates whether to reload the page (`true`) or not (`false`).

Plug-in Side Plug-in API

This chapter describes methods in the plug-in API that are available for the `plug-in` object. The names of all of these methods begin with `NPP_` to indicate that they are implemented by the plug-in and called by the browser. For an overview of how these two sides of the plug-in API interact, see the ***How Plug-ins Work*** and ***Overview of Plug-in Structure*** sections in the introduction.

Plugin Method Summary

NPP_Destroy	Deletes a specific instance of a plug-in.
NPP_DestroyStream	Tells the plug-in that a stream is about to be closed or destroyed.
NPP_GetValue	Allows the browser to query the plug-in for information.
NPP_HandleEvent	Delivers a platform-specific window event to the instance.
NP_Initialize	Provides global initialization for a plug-in.
NPP_New	Creates a new instance of a plug-in.
NPP_NewStream	Notifies a plug-in instance of a new data stream.
NPP_Print	Requests a platform-specific print operation for an embedded or full-screen plug-in.
NPP_SetValue	Sets information about the plug-in.
NPP_SetWindow	Tells the plug-in when a window is created, moved, sized, or destroyed.
NP_Shutdown	Provides global deinitialization for a plug-in.

NPP_StreamAsFile	Provides a local file name for the data from a stream.
NPP_URLNotify	Notifies the instance of the completion of a URL request.
NPP_Write	Delivers data to a plug-in instance.
NPP_WriteReady	Determines maximum number of bytes that the plug-in can consume.

NPP_Destroy

Deletes a specific instance of a plug-in.

Syntax

```
#include <npapi.h>
NPError NPP_Destroy(NPP instance, NPSavedData **save);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the plug-in instance to delete.
`**save`	State or other information to save for reuse by a new instance of this plug-in at the same URL. Passed to `NPP_New`.

Returns

If successful, the function returns `NPERR_NO_ERROR`.

If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

NPP_Destroy releases the instance data and resources associated with a plug-in. The browser calls this function when a plug-in instance is deleted, typically because the user has left the page containing the instance, closed the window, or quit the browser. You should delete any private instance-specific information stored in the plug-in's instance->pdata at this time.

If this function is deleting the last instance of a plug-in, NP_Shutdown is subsequently called. Use **NP_Shutdown** to delete any data allocated in **NP_Initialize** and intended to be shared by all instances of a plug-in.

Use the optional save parameter if you want to save and reuse some state or other information. Upon the user's return to the page, this information is passed to the new plug-in instance when it is created with **NPP_New**.

Avoid trying to save critical data with this function. Ownership of the buf field of the **NPSavedData** structure passes from the plug-in to the browser when NPP_Destroy returns. The browser can and will discard this data based on arbitrary criteria such as its size and the user's page history.

To ensure that the browser does not crash or leak memory when the saved data is discarded, NPSavedData's buf field should be a flat structure (a simple structure with no allocated substructures) allocated with **NPN_MemAlloc**.

Mac OS

If you want to restore state information if this plug-in is later recreated, use NP_MemAlloc to create an NPSavedData structure. §

NOTE: You should not perform any graphics operations in NPP_Destroy as the instance's window is no longer guaranteed to be valid. §

See Also

NPP_New, NP_Shutdown, NPP, NPN_MemAlloc, NPSavedData,

NPP_DestroyStream

Tells the plug-in that a stream is about to be closed or destroyed.

Syntax

```
#include <npapi.h>
NPError NPP_DestroyStream(NPP       instance,
                          NPStream* stream,
                          NPReason  reason);
```

Parameters

The function has the following parameters:

`instance`	Pointer to current plug-in instance.
`stream`	Pointer to current stream.
`reason`	Reason the stream was destroyed. Values: `NPRES_DONE` (Most common): Completed normally; all data was sent to the instance. `NPRES_USER_BREAK`: User canceled stream directly by clicking the Stop button or indirectly by some action such as deleting the instance or initiating higher-priority network operations. `NPRES_NETWORK_ERR`: Stream failed due to problems with network, disk I/O, lack of memory, or other problems.

Returns

If successful, the function returns `NPERR_NO_ERROR`.

If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

The browser calls the `NPP_DestroyStream` function when a data stream sent to the plug-in is finished, either because it has completed successfully or terminated abnormally. After this, the browser deletes the **NPStream** object.

You should delete any private data allocated in stream->pdata at this time, and should not make any further references to the stream object.

See Also

`NPP_NewStream`, `NPP_DestroyStream`, `NPStream`

NPP_GetValue

Allows the browser to query the plug-in for information.

Syntax

```
#include <npapi.h>
NPError NPP_GetValue(void *instance,
                     NPPVariable variable,
                     void *value);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the current plug-in instance.
`variable`	Unix only: Plug-in information the call gets. Values: `NPPVpluginNameString`: Gets the name of the plug-in `NPPVpluginDescriptionString`: Gets the description string of the plug-in `NPPVpluginWindowBool`: Tells whether the plug-in is windowless; true=windowless, false=not windowless `NPPVpluginTransparentBool`: Tells whether the plug-in is transparent; true=transparent, false=not transparent
`value`	Plug-in name, returned by the function.

Returns

If successful, the function returns `NPERR_NO_ERROR`.

If unsuccessful, the function returns an error code. For possible values, see **Error Codes**.

Description

`NPP_GetValue` retrieves plug-in features set with **NPP_SetValue**, among them whether a plug-in is windowed or windowless and whether JavaScript is enabled.

You can use this method as an optional entry point that the browser can call to determine the plug-in name and description. It returns the requested values, specified by the variable and value parameters, to the plug-in.

See Also

`NPP_SetValue`

NPP_HandleEvent

Delivers a platform-specific window event to the instance.

For Windowed Plug-ins: Currently used only on Mac OS.

For Windowless Plug-ins: Windows and Mac OS.

Syntax

```
#include <npapi.h>
int16 NPP_HandleEvent(NPP instance, void* event);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the current plug-in instance.
`event`	Platform-specific value representing the event handled by the function. Values: MS Windows: Pointer to `NPEvent` structure Mac OS: Pointer to a standard Mac OS `EventRecord` For a list of possible events for MS Windows and Mac OS, see NPEvent.

Returns

If the plug-in handles the event, the function should return true.

If the plug-in ignores the event, the function returns false.

Description

The browser calls **NPP_HandleEvent** to tell the plug-in when events take place in the plug-in's window or drawable area. The plug-in either handles or ignores the event, depending on the value given in the event parameter of this function. For a list of event types the application is responsible for delivering to the plug-in, see the **NPEvent** structure.

MS Windows

The browser gives each windowed plug-in its own native window, often a child window of the browser window, to draw into. The plug-in has complete control over drawing and event handling within that window. §

Mac OS

The browser does not give a windowed plug-in a native window, because the Mac OS platform does not support child windows. Instead, the windowed plug-in draws into the graphics port associated with the the browser window, at the offset that the browser specifies. For this reason, **NPP_HandleEvent** is only way the plug-in can receive events from its host application on Mac OS. When **NPP_HandleEvent** is called, the current port is set up so that its origin matches the top-left corner of the plug-in. A plug-in does not need to set up the current port for mouse coordinate translation. §

See Also

`NPEvent`

NP_Initialize

Provides global initialization for a plug-in.

Syntax

```
#include <npapi.h>
NPError NP_Initialize(void)
```

Returns

If successful, the function returns `NPERR_NO_ERROR`.

If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

The browser calls this function only once: when a plug-in is loaded, before the first instance is created. This is the first function that the browser calls. **NP_Initialize** tells the plug-in that the browser has loaded it and provides global initialization. Allocate any memory or resources shared by all instances of your plug-in at this time.

After the last instance of a plug-in has been deleted, the browser calls **NP_Shutdown**, where you can release allocated memory or resources.

MS Windows

See Also

`NP_Shutdown, NPP_New`

NPP_New

Creates a new instance of a plug-in.

Syntax

```
#include <npapi.h>
NPError NPP_New(NPMIMEType    pluginType,
                NPP instance, uint16 mode,
                int16 argc,   char *argn[],
                char *argv[], NPSavedData *saved);
```

Parameters

The function has the following parameters:

`pluginType`	Pointer to the MIME type for new plug-in instance.
`instance`	Contains instance-specific private data for the plug-in and the browser. This data is stored in instance->pdata.
`mode`	Display mode of plug-in. Values: • `NP_EMBED`: (1) Instance was created by an `EMBED` tag and shares the browser window with other content. • `NP_FULL`: (2) Instance was created by a separate file and is the primary content in the window.
`argc`	Number of HTML arguments in the `EMBED` tag for an embedded plug-in; determines the number of attributes in the argn and argv arrays.
`argn[]`	Array of attribute names passed to the plug-in from the `EMBED` tag.
`argv[]`	Array of attribute values passed to the plug-in from the `EMBED` tag.
`saved`	Pointer to data saved by **`NPP_Destroy`** for a previous instance of this plug-in at the same URL. If non-null, the browser passes ownership of the `NPSavedData` object back to the plug-in. The plug-in is responsible for freeing the memory for the `NPSavedData` and the buffer it contains.

Returns

- If successful, the function returns `NPERR_NO_ERROR`.
- If unsuccessful, the function returns an error code. For possible values, see **Error Codes**.

Description

NPP_New creates a new instance of a plug-in. It is called after **NP_Initialize** and provides the MIME type, embedded or full-screen display mode, and, for embedded plug-ins, information about HTML EMBED arguments.

The plug-in's NPP pointer is valid until the instance is destroyed with **NPP_Destroy**.

If instance data was saved from a previous instance of the plug-in by the **NPP_Destroy** function, it is returned in the saved parameter for the current instance to use.

All attributes in the EMBED tag (standard and private) are passed in NPP_New in the argn and argv arrays. The browser ignores any non-standard attributes within an EMBED tag. This gives developers a chance to use private attributes to communicate instance-specific options or other information to the plug-in. Place private options at the end of the list of standard attributes in the EMBED Tag.

See Also

NPP_Destroy, **NP_Shutdown**, **NPP**, **NPSavedData**

NPP_NewStream

Notifies a plug-in instance of a new data stream.

Syntax

```
#include <npapi.h>
NPError NPP_NewStream(NPP       instance,
                      NPMIMEType type,
                      NPStream   *stream,
                      NPBool     seekable,
                      uint16*    stype);
```

Parameters

The function has the following parameters:

`instance`	Pointer to current plug-in instance.
`type`	Pointer to MIME type of the stream.
`stream`	Pointer to new stream.
`seekable`	Boolean indicating whether the stream is seekable: `true`: Seekable. Stream supports random access through calls to **NPN_RequestRead** (for example, local files or HTTP servers that support byte-range requests). `false`: Not seekable. The browser must copy data in the stream to the local cache to satisfy random access requests made through **NPN_RequestRead**.
`stype`	Requested mode of new stream. For more information about each of these values, see Directions in this section. `NP_NORMAL` (Default): Delivers stream data to the instance in a series of calls to NPP_WriteReady and NPP_Write. `NP_ASFILEONLY`: Saves stream data to a file in the local cache. `NP_ASFILE`: File download. Like `NP_ASFILEONLY` except that data is delivered to the plug-in as it is saved to the file (as in mode `NP_NORMAL`). `NP_SEEK`: Stream data randomly accessible by the plug-in as needed, through calls to **NPN_RequestRead**.

Returns

If successful, the function returns NPERR_NO_ERROR.

If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

NPP_NewStream notifies the plug-in when a new stream is created. The NPStream* pointer is valid until the stream is destroyed. The plug-in can store plug-in-private data associated with the stream in stream->pdata. The MIME type of the stream is provided by the type parameter.

The data in the stream can be the file specified in the SRC attribute of the EMBED tag, for an embedded instance, or the file itself, for a full-page instance. A plug-in can also request a stream with the function **NPN_GetURL**. The browser calls **NPP_DestroyStream** when the stream completes (either successfully or abnormally). The plug-in can terminate the stream itself by calling **NPN_DestroyStream**.

The parameter stype defines the mode of the stream. Values:

- NP_NORMAL (Default): Delivers stream data to the instance in a series of calls to **NPP_WriteReady** and **NPP_Write**. The plug-in can process the data progressively as it arrives from the network or file system.
- NP_ASFILEONLY: The browser saves stream data to a file in the local cache. When the stream is complete, the browser calls NPP_StreamAsFile to deliver the path of the file to the plug-in. If the stream comes from a local file, the **NPP_Write** and **NPP_WriteReady** functions are not called. **NPP_StreamAsFile** is simply called immediately. This mode allows the plug-in full random access to the data using platform-specific file operations.
- NP_ASFILE: File download. Differs from NP_ASFILEONLY in that data is delivered to the plug-in, through a series of calls to **NPP_WriteReady** and **NPP_Write**, as it is saved to the file (as in mode NP_NORMAL). When the stream is complete, the browser calls **NPP_StreamAsFile** to deliver the path of the file to the plug-in. If the data in the stream comes from a file that is already local, the data is read, sent to the plug-in through NPP_Write, and written to a file in the local cache.

  NOTE: Most plug-ins that need the stream saved to a file should use the more efficient mode NP_ASFILEONLY (above); this mode is preserved for compatibility only.
- NP_SEEK: Stream data is not automatically delivered to the instance, but can be randomly accessed by the plug-in as needed, through calls to

NPN_RequestRead. If the stream is not seekable, placing the stream in `NP_SEEK` mode causes the browser to save the entire stream to the disk cache. **NPN_RequestRead** requests are only fulfilled when all data has been read and stored in the cache. As an optimization to extract the maximum benefit from existing network connections, the browser continues to read data sequentially out of the stream (as in mode `NP_NORMAL`) until the first **NPN_RequestRead** call is made.

NOTE: In any mode other than `NP_SEEK`, the application should call **NPP_DestroyStream** once all data in the stream has been written to the plug-in. The plug-in can also request termination of the stream at any time by calling **NPP_DestroyStream**. §

See Also

`NPN_NewStream`, **`NPP_StreamAsFile`**, **`NPP_Write`**, **`NPP_WriteReady`**, **`NPP_DestroyStream`**, **`NPN_RequestRead`**, **`NPStream`**, **`NPN_GetURL`**

NPP_Print

Requests a platform-specific print operation for an embedded or full-screen plug-in.

Syntax

```
#include <npapi.h>
void NPP_Print(NPP instance, NPPrint* PrintInfo);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the current plug-in instance. Must be embedded or full-screen.
`printInfo`	Pointer to `NPPrint` structure.

Description

`NPP_Print` is called when the user requests printing for a web page that contains a visible plug-in (either embedded or full-page). It uses the print mode set in the **NPPrint** structure in its printInfo parameter to determine whether the plug-in should print as an embedded plug-in or as a full-page plug-in.

- An embedded plug-in shares printing with the browser; the plug-in prints the part of the page it occupies, and the browser handles everything else, including displaying print dialog boxes, getting the printer device context, and any other tasks involved in printing, as well as printing the rest of the page. For an embedded plug-in, set the printInfo field to **NPEmbedPrint**.
- A full-page plug-in handles all aspects of printing itself. For a full-page plug-in, set the printInfo field to `NPFullPrint` or null.

For information about printing on your platform, see your platform documentation.

MS Windows

On MS Windows, `printInfo->print.embedPrint.platformPrint` is the device context (DC) handle. Be sure to cast this to type `HDC`. §

The coordinates for the window rectangle are in TWIPS format. This means that you need to convert the x-y coordinates using the Windows API call `DPtoLP` when you output text. §

See Also

`NPPrint`, `NPFullPrint`, `NPEmbedPrint`

NPP_SetValue

Sets information about the plug-in.

Syntax

```
#include <npapi.h>
NPError NPP_SetValue(void *instance,
                     NPPVariable variable,
                     void *value);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the current plug-in instance.
`variable`	The plug-in information the call is setting. For values, see `NPP_GetValue`.
`value`	Destination for plug-in information returned by the function.

Returns

If successful, the function returns `NPERR_NO_ERROR`.

If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

`NPP_SetValue` sets a variety of features for a plug-in, among them whether a plug-in is windowed or windowless and whether JavaScript is enabled. For possible values, see **NPP_GetValue**. The plug-in makes this call from its **NPP_New** method.

For example, to specify that a plug-in is windowless, use `NPP_SetValue` with `NPPVpluginWindowBool` as the variable to set and false as the value parameter. If a plug-in does not make this call, it is considered a windowed plug-in.

See Also

`NPP_New`, `NPP_GetValue`

NPP_SetWindow

Tells the plug-in when a window is created, moved, sized, or destroyed.

Syntax

```
#include <npapi.h>
NPError NPP_SetWindow(NPP instance, NPWindow *window);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the current plug-in instance. Must be embedded or full-screen.
`window`	Pointer to the window into which the instance draws. The window structure contains a window handle and values for top left corner, width, height, and clipping rectangle (see note on Unix below).

Returns

If successful, the function returns `NPERR_NO_ERROR`.

If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

The browser calls `NPP_SetWindow` after creating the instance to allow drawing to begin. Subsequent calls to `NPP_SetWindow` indicate changes in size or position; these calls pass the same **NPWindow** object each time, but with different values. If the window handle is set to null, the window is destroyed. In this case, the plug-in must not perform any additional graphics operations on the window and should free any associated resources.

The data structure passed in `NPP_SetWindow` is an **NPWindow** object, which contains the coordinates of the instance's area and various platform-specific data. This window is valid for the life of the instance, or until `NPP_SetWindow` is called again with a different value.

For windowed plug-ins on Windows and Unix, the window parameter contains a handle to a subwindow of the browser window hierarchy. On Mac OS, this field points to an **NP_Port** structure. For windowless plug-ins, it is a platform-specific handle to a drawable.

Before setting the window parameter to point to a new window, it is a good idea to compare the information about the new window to the previous window (if one existed) to account for any changes.

NOTE: `NPP_SetWindow` is useful only for embedded (`NP_EMBED`) or full-screen (`NP_FULL`) plug-ins, which are drawn into windows. It is irrelevant for hidden plug-ins.§

See Also

`NPP_HandleEvent`, `NPWindow`, `NP_Port`

NP_Shutdown

Provides global deinitialization for a plug-in.

Syntax

```
#include <npapi.h>
void NP_Shutdown(void);
```

Description

The browser calls this function once after the last instance of your plug-in is destroyed, before unloading the plug-in library itself. Use `NP_Shutdown` to delete any data allocated in **NP_Initialize** to be shared by all instances of a plug-in.

If you have defined a Java class for your plug-in, be sure to release it at this time so that Java can unload it and free up memory.

NOTE: If enough memory is available, the browser can keep the plug-in library loaded if it expects to create more instances in the near future. The browser calls `NP_Shutdown` only when the library is finally unloaded. §

MS Windows

See Also

`NP_Initialize`, **`NPP_Destroy`**

NPP_StreamAsFile

Provides a local file name for the data from a stream.

Syntax

```
#include <npapi.h>
void NPP_StreamAsFile(NPP        instance,
                      NPStream*  stream,
                      const char* fname);
```

Parameters

The function has the following parameters:

`instance`	Pointer to current plug-in instance.
`stream`	Pointer to current stream.
`fname`	Pointer to full path to a local file. If an error occurs while retrieving the data or writing the file, fname may be null.

Description

When the stream is complete, the browser calls `NPP_StreamAsFile` to provide the instance with a full path name for a local file for the stream. `NPP_StreamAsFile` is called for streams whose mode is set to `NP_ASFILEONLY` or `NP_ASFILE` only in a previous call to **NPP_NewStream**.

If an error occurs while retrieving the data or writing the file, the file name (fname) is null.

See Also

`NPP_NewStream`, **`NPP_Write`**, **`NPP_WriteReady`**, **`NPStream`**, **`NPP`**

NPP_URLNotify

Notifies the instance of the completion of a URL request.

Syntax

```
#include <npapi.h>
void NPP_URLNotify(NPP         instance,
                   const char* url,
                   NPReason    reason,
                   void*       notifyData);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the current plug-in instance.
`url`	URL of the `NPN_GetURLNotify` or `NPN_PostURLNotify` request.
`reason`	Reason code for completion of request. Values: • `NPRES_DONE` (most common): Completed normally. • `NPRES_USER_BREAK`: User canceled stream directly by clicking the Stop button or indirectly by some action such as deleting the instance or initiating higher-priority network operations. • `NPRES_NETWORK_ERR`: Stream failed due to problems with network, disk I/O, lack of memory, or other problems.
`notifyData`	Plug-in-private value for associating a previous `NPN_GetURLNotify` or `NPN_PostURLNotify` request with a subsequent `NPP_URLNotify` call.

Description

The browser calls `NPP_URLNotify` after the completion of a **NPN_GetURLNotify** or **NPN_PostURLNotify** request to inform the plug-in that the request was completed and supply a reason code for the completion.

The most common reason code is `NPRES_DONE`, indicating simply that the request completed normally. Other possible reason codes are `NPRES_USER_BREAK`, indicating that the request was halted due to a user action (for example, clicking the Stop button), and `NPRES_NETWORK_ERR`, indicating that the request could not be completed, perhaps because the URL could not be found.

The parameter `notifyData` is the plug-in-private value passed as an argument by a previous **NPN_GetURLNotify** or **NPN_PostURLNotify** call, and can be used as an identifier for the request.

See Also

`NPN_GetURLNotify`, **`NPN_GetURL`**, **`NPN_PostURLNotify`**, **`NPN_PostURL`**

NPP_Write

Delivers data to a plug-in instance.

Syntax

```
#include <npapi.h>
int32 NPP_Write(NPP instance,
                NPStream* stream,
                int32 offset,
                int32 len,
                void* buf);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the current plug-in instance.
`stream`	Pointer to the current stream.
`offset`	Offset in bytes of buf from the beginning of the data in the stream. Can be used to check stream progress or bye range requests from `NPN_RequestRead`.
`len`	Length in bytes of buf; number of bytes accepted.
`buf`	Buffer of data, delivered by the stream, that contains len bytes of data offset bytes from the start of the stream. The buffer is allocated by the browser and is deleted after returning from the function, so the plug-in should make a copy of the data it needs to keep.

Returns

If successful, the function returns the number of bytes consumed by the instance.

If unsuccessful, the function destroys the stream by returning a negative value.

Description

The browser calls the `NPP_Write` function to deliver the data specified in a previous **NPP_WriteReady** call to the plug-in. A plug-in must consume at least as many bytes as indicated in the **NPP_WriteReady** call.

After a stream is created by a call to **NPP_NewStream**, the browser calls `NPP_Write` either:

- If the plug-in requested a normal-mode stream, the data in the stream is delivered to the plug-in instance in a series of calls to **NPP_WriteReady** and `NPP_Write`.
- If the plug-in requested a seekable stream, the **NPN_RequestRead** function requests reads of a specified byte range that results in a series of calls to **NPP_WriteReady** and `NPP_Write`.

The plug-in can use the offset parameter to track the bytes that are written. This gives you different information depending in the type of stream. In a normal-mode stream., the parameter value increases as the each buffer is written. The `buf` parameter is not persistent, so the plug-in must process data immediately or allocate memory and save a copy of it. In a seekable stream with byte range requests, you can use this parameter to track **NPN_RequestRead** requests.

The plug-in should return the number of bytes written (consumed by the instance). If the return value is smaller than the size of the buffer, the browser sends the remaining data to the plug-in through subsequent calls to **NPP_WriteReady** and `NPP_Write`. A negative return value causes an error on the stream, which causes the browser to destroy the stream with **NPP_DestroyStream**.

See Also

`NPP_DestroyStream`, `NPP_NewStream`, `NPP_WriteReady`, `NPStream`, `NPP`

NPP_WriteReady

Determines maximum number of bytes that the plug-in can consume.

Syntax

```
#include <npapi.h>
int32 NPP_WriteReady(NPP instance, NPStream* stream);
```

Parameters

The function has the following parameters:

instance	Pointer to the current plug-in instance.
stream	Pointer to the current stream.

Returns

Returns the maximum number of bytes that an instance is prepared to accept from the stream.

Description

The browser calls `NPP_WriteReady` before each call to **NPP_Write** to determine whether a plug-in can receive data and how many bytes it can receive. This function allows the browser to send only as much data to the instance as it can handle at one time, making resource use more efficient for both the browser and plug-in.

The `NPP_Write` function may pass a larger buffer, but the plug-in is required to consume only the amount of data returned by `NPP_WriteReady`.

The browser can write a smaller amount of data if desired or necessary; for example, if only 8K of data is available in a network buffer. If the plug-in is allocating memory for the entire stream at once (an `AS_FILE` stream), it can return a very large number. Because it is not processing streaming data, the browser can pass as much data to the instance as necessary in a single **NPP_Write**.

If the plug-in receives a value of zero, the data flow temporarily stops. The browser checks to see if the plug-in can receive data again by resending the data at regular intervals.

See Also

`NPP_Write`, **`NPStream`**, **`NPP`**

Browser Side Plug-in API

This chapter describes methods in the plug-in API that are available for the `browser`. The names of all of these methods begin with `NPN_` to indicate that they are implemented by the browser and called by the plug-in. For an overview of how these two sides of the plug-in API interact, see the ***How Plug-ins Work*** and ***Overview of Plug-in Structure*** sections in the introduction.

Netscape Plug-in Method Summary

NPN_DestroyStream	Closes and deletes a stream.
NPN_ForceRedraw	Forces a paint message for a windowless plug-in.
NPN_GetURL	Asks the browser to create a stream for the specified URL.
NPN_GetURLNotify	Requests creation of a new stream with the contents of the specified URL; gets notification of the result.
NPN_GetValue	Allows the plug-in to query the browser for information.
NPN_InvalidateRect	Invalidates specified drawing area prior to repainting or refreshing a windowless plug-in.
NPN_InvalidateRegion	Invalidates specified drawing region prior to repainting or refreshing a windowless plug-in.
NPN_MemAlloc	Allocates memory from the browser's memory space.
NPN_MemFlush	Requests that the browser free a specified amount of memory.
NPN_MemFree	Deallocates a block of allocated memory.

NPN_NewStream	Requests the creation of a new data stream produced by the plug-in and consumed by the browser.
NPN_PostURL	Posts data to a URL.
NPN_PostURLNotify	Posts data to a URL, and receives notification of the result.
NPN_ReloadPlugins	Reloads all plug-ins in the Plugins directory.
NPN_RequestRead	Requests a range of bytes for a seekable stream.
NPN_SetValue	Sets windowless plug-in as transparent or opaque.
NPN_Status	Displays a message on the status line of the browser window.
NPN_UserAgent	Returns the browser's user agent field.
NPN_Version	Returns version information for the Plug-in API.
NPN_Write	Pushes data into a stream produced by the plug-in and consumed by the browser.

NPN_DestroyStream

Closes and deletes a stream.

Syntax

```
#include <npapi.h>
NPError NPN_DestroyStream(NPP      instance,
                          NPStream* stream,
                          NPError   reason);
```

Parameters

The function has the following parameters:

`instance`	Pointer to current plug-in instance.
`stream`	Pointer to current stream, initiated by either the browser or the plug-in.
`reason`	Reason the stream was stopped so the application can give the user appropriate feedback. Values: • `NPRES_DONE` (most common): Stream completed normally; all data was sent by the plug-in to the browser. • `NPRES_USER_BREAK`: Plug-in is terminating the stream due to a user request. • `NPRES_NETWORK_ERR`: Stream failed due to network problems.

Returns

If successful, the function returns `NPERR_NO_ERROR`.

If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

The plug-in calls the `NPN_DestroyStream` function to close and delete a stream. This stream can be either a stream that the browser created and passed to the plug-in in **`NPP_NewStream`**, or a stream created by the plug-in through a call to `NPN_NewStream`.

See Also

`NPP_DestroyStream`, **`NPN_NewStream`**, **`NPStream`**, **`NPP`**

NPN_ForceRedraw

Forces a paint message for a windowless plug-in.

Syntax

```
#include <npapi.h>
void NPN_ForceRedraw(NPP instance);
```

Parameters

The function has the following parameters:

`instance`	Plug-in instance for which the function forces redrawing.

Description

A windowed plug-in determines when it draws, while a windowless plug-in draws only in response to a paint message from the browser. **`NPN_ForceRedraw`** forces a paint message for a windowless plug-in.

Once a value has been invalidated with **NPN_InvalidateRect** or **NPN_InvalidateRegion**, a plug-in can call `NPN_ForceRedraw` to force a paint message. This causes a synchronous update event or paint message for the plug-in.

MS Windows

The plug-in receives a `WM_PAINT` message. The `lParam` of the `WM_PAINT` message holds a pointer to an `NPRect` that is the bounding box of the update area. Since the plug-in and the browser share the same HDC, before drawing, the plug-in is responsible for saving the current HDC settings, setting up its own environment, drawing, and restoring the HDC to the previous settings. The HDC settings must be restored whenever control returns back to the browser, either before returning from `NPP_HandleEvent` or before calling a drawing-related Netscape method. §

Mac OS

The plug-in receives an updateEvent. The clipRegion of the drawable's CGrafPtr is set to the update region. As is the case for windowed plug-ins on Mac OS, the plug-in must first save the current settings of the port, setting up the drawing environment as appropriate, drawing, and restoring the port to the previous setting. This should happen before the plug-in returns from `NP_HandleEvent` or before the plug-in calls a drawing-related Navigator method. §

See Also

NPN_InvalidateRect, **NPN_InvalidateRegion**, **NPP**

NPN_GetURL

Asks the browser to create a stream for the specified URL.

Syntax

```
#include <npapi.h>
NPError NPN_GetURL(NPP instance,
                   const char* url,
                   const char* target);
```

Parameters

The function has the following parameters:

`instance` — Pointer to the current plug-in instance.

`url` — Pointer to the URL of the request. Can be of any type, such as HTTP, FTP, news, mailto, gopher.

`target` — Name of the target window or frame, or one of the following special target names. Values:

- `_blank` or `_new`: Load the link in a new blank unnamed window. Safest target, even though, when used with a mailto or news URL, this creates an extra blank the browser instance.
- `_self` or `_current`: Load the link into the same window the plug-in instance occupies. Not recommended; see Warning. If target refers to the window or frame containing the instance, the instance is destroyed and the plug-in may be unloaded. Use with `NPN_GetURL` only if you want to terminate the plug-in.
- `_parent`: Load the link into the immediate `FRAMESET` parent of the plug-in instance's document. If the plug-in instance's document has no parent, the default is `_self`.
- `_top`: Load the link into the plug-in instance window. The default is `_self`, if the plug-in instance's document is already at the top. Use for breaking out of a deep frame nesting.

If null, the browser creates a new stream and delivers the data to the current instance regardless of the MIME type of the URL. In general, if a URL works in the location box of the Navigator, it works here, except for the `_self` target.

Returns

- If successful, the function returns `NPERR_NO_ERROR`.

- If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

`NPN_GetURL` is used to load a URL into the current window or another target or stream. Plug-ins can use this capability to provide hyperlinks to other documents or to retrieve data from anywhere on the network. This is especially useful for enabling an existing application to operate on the web.

For HTTP URLs, the browser resolves this method as the HTTP server method `GET`, which requests URL objects.

Use **NPN_PostURLNotify** instead of **NPN_PostURL** in these cases:
- To request a stream and receive notification of the result.
- If the buffer contains header information (even a blank line).

Make sure that the target matches the URL type sent to it. For example, a null target does not make sense for some URL types (such as mailto). The following recommendations about target choice apply to other methods that handle URLs as well.

If the target parameter refers to the window or frame containing the current plug-in instance, the instance is destroyed and the plug-in may be unloaded. If target is null, the application creates a new stream and delivers the data to the plug-in instance, through calls to **NPP_NewStream**, **NPP_WriteReady** and **NPP_Write**, and **NPP_DestroyStream**. This means that if you want the plug-in to handle a new stream, no matter what the MIME type is, use null. If the application cannot locate the URL and retrieve the data, it does not create a stream for the instance.

When the plug-in instance is part of a regular Navigator window, and it uses a `_blank` target with a mailto or news URL, another blank navigator window is opened along with the mail or news window.

When the plug-in uses a `_self` target, no other instance is created; the plug-in usually continues to operate successfully in its own window. The safest target is `_blank`, even though this creates an extra blank the browser instance.

For complete information on named targets for this function (as well as for normal HTML links), see the Netscape document, "Targeting Windows."

The plug-in developer cannot influence the way that the browser handles `NPN_GetURL`. It is typically asynchronous but this is not guaranteed. The plug-in could call `NPN_GetURL` and receive data from the URL right away, but more often the data arrives later. The rest of the the browser interface keeps running until the data is available. §

See Also

`NPN_GetURLNotify`, **`NPN_PostURL`**, **`NPN_PostURLNotify`**, **`NPP_URLNotify`**

NPN_GetURLNotify

Requests creation of a new stream with the contents of the specified URL; gets notification of the result.

Syntax

```
#include <npapi.h>
NPError NPN_GetURLNotify(NPP    instance,
                         const  char* url,
                         const  char* target,
                         void*   notifyData);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the current plug-in instance.
`url`	Pointer to the URL of the request.
`target`	Name of the target window or frame, or one of several special target names. For values, see **NPN_GetURL**.
`notifyData`	Plug-in-private value for associating the request with the subsequent **NPP_URLNotify** call, which passes this value (see Description below).

Returns

- If successful, the function returns `NPERR_NO_ERROR`.
- If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

`NPN_GetURLNotify` works just like **NPN_GetURL**, with one exception. `NPN_GetURLNotify` notifies the plug-in instance upon successful or unsuccessful completion of the request by calling the plug-in's **NPP_URLNotify** function and passing it the `notifyData` value.

`NPN_GetURLNotify` typically handles the URL request asynchronously. It returns immediately and only later handles the request and calls `NPP_URLNotify`. This notification is the only way the plug-in can tell whether a request with a null target failed, or that a request with a non-null target completed.

For requests that complete unsuccessfully, the browser calls `NPP_URLNotify` as soon as possible. For requests that complete successfully:

- If the target is non-null, the browser calls `NPP_URLNotify` after it has finished loading the URL.
- If the target is null, the browser calls `NPP_URLNotify` after closing the stream by calling **NPN_DestroyStream**.

If this function is called with a target parameter value of `_self` or a parent to `_self`, this function should return an `INVALID_PARAM` NPError. This is the only way to notify the plug-in once it is deleted.

See Also

`NPN_GetURL`, `NPN_PostURL`, `NPN_PostURLNotify`, `NPP_URLNotify`, `NPP`

NPN_GetValue

Allows the plug-in to query the browser for information.

Syntax

```
#include <npapi.h>
NPError NPN_GetValue(NPP         instance,
                     NPNVariable variable,
                     void        *value);
```

Parameters

This function has the following parameters:

`instance` — Pointer to the current plug-in instance.

`variable` — Information the call gets. Values for `NPNVariable`:

- `NPNVxDisplay =1`: Unix only: Returns the current Display
- `NPNVxtAppContext`: Unix only: Returns the application's `XtAppContext`
- `NPNVnetscapeWindow`: MS Windows only: Gets the native window on which plug-in drawing occurs; returns HWND
- `NPNVjavascriptEnabledBool`: Tells whether JavaScript is enabled; true=JavaScript enabled, false=not enabled
- `NPNVasdEnabledBool`: Tells whether SmartUpdate (former name: ASD) is enabled; true=SmartUpdate enabled, false=not enabled
- `NPNVOfflineBool`: Tells whether offline mode is enabled; true=offline mode enabled, false=not enabled

`value` — Function returns the name of the plug-in in the value parameter.

Returns

- If successful, the function returns `NPERR_NO_ERROR`.
- If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

NPN_GetValue returns the browser information set with **NPN_SetValue**. The queried information is returned in the value parameter.

The method returns a value of type HWND. In many cases, a plug-in may still have to create its own window (a transparent child window of the browser window) to act as the owner window for popup menus and modal dialogs. This transparent child window can have its own WindowProc within which the plug-in can deal with WM_COMMAND messages sent to it a result of tracking the popup menu or modal dialog.

Unix

The values for this parameter are the NPNVxDisplay (the current Display) and the NPNVxtAppContext (the browser's XtAppContext). §

MS Windows

You can use this method to help create a menu or dialog box for a windowless plug-in. In order to bring up popup menus and modal dialogs, a plug-in needs a parent window. A windowless plug-in does not receive its own native window. Instead, it draws directly into the drawable given to it. Use the NPNVnetscapeWindow value to get the native window on which plug-in drawing occurs. §

See Also

NPN_SetValue, **NPP_GetValue**, **NPN_SetValue**

NPN_InvalidateRect

Invalidates specified drawing area prior to repainting or refreshing a windowless plug-in.

Syntax

```
#include <npapi.h>
void NPN_InvalidateRect(NPP instance,
                        NP_Rect *invalidRect);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the current plug-in instance.
`invalidRect`	The area to invalidate, specified in a coordinate system that originates at the top left of the plug-in.

Description

Before a windowless plug-in can repaint or refresh part of its drawing area, the plug-in must first invalidate the area with either `NPN_InvalidateRect` or **NPN_InvalidateRegion**.

`NPN_InvalidateRect` causes the **NPP_HandleEvent** method to pass an update event or a paint message to the plug-in. After calling this method, the plug-in receives a paint message asynchronously.

The browser redraws invalid areas of the document and any windowless plug-ins at regularly timed intervals. To force a paint message, the plug-in can call **NPN_ForceRedraw** after calling this method.

See Also

`NPN_ForceRedraw`, **`NPN_InvalidateRegion`**, **`NP_Rect`**, **`NPP`**

NPN_InvalidateRegion

Invalidates specified drawing region prior to repainting or refreshing a windowless plug-in.

Syntax

```
#include <npapi.h>
void NPN_InvalidateRegion(NPP instance,
                          NP_Region invalidRegion);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the current plug-in instance.
`invalidRegion`	The area to invalidate, specified in a coordinate system that originates at the top left of the plug-in.

Description

Before a windowless plug-in can repaint or refresh part of its drawing area, the plug-in must first invalidate the area with either **NPN_InvalidateRect** or `NPN_InvalidateRegion`.

`NPN_InvalidateRegion` causes the **NPP_HandleEvent** method to pass an update event or a paint message to the plug-in. If a plug-in calls this method, it receives a paint message later. The browser redraws invalid areas of the document and windowless plug-ins at regularly timed intervals. To force a paint message, the plug-in can call **NPN_ForceRedraw** after calling this method.

See Also

`NPN_ForceRedraw`, `NPN_InvalidateRect`, `NP_Region`, `NPP`

NPN_MemAlloc

Allocates memory from the browser's memory space.

Syntax

```
#include <npapi.h>
void *NPN_MemAlloc (uint32 size);
```

Parameters

The function has the following parameters:

`size`	Size of memory, in bytes, to allocate in the browser's memory space.

Returns

- If successful, the function returns a pointer to the allocated memory, in bytes.
- If insufficient memory is available, the plug-in returns null.

Description

The plug-in calls `NPN_MemAlloc` to allocate a specified amount of memory in the browser's memory space. If you allocate saved instance data with **NPP_Destroy**, be sure to use `NPN_MemAlloc` to allocate memory. This ensures that the browser can free the saved data at a later time with the equivalent of **NPN_MemFree**.

Since the browser and plug-ins share the same memory space, `NPN_MemAlloc` allows plug-ins to take advantage of any customized memory allocation scheme the application may have, and allows the application to manage its memory more flexibly and efficiently.

Mac OS

`NPN_MemAlloc` is particularly important on Mac OS, since the Mac OS version of the browser frequently fills its memory partition with cached data that is only purged as necessary. Since `NPN_MemAlloc` automatically frees cached information if necessary to fulfill the request, calls to `NPN_MemAlloc` may succeed where direct calls to `NewPtr` fail. §

Mac OS

Existing calls to `NPN_MemFlush` have no effect. You only need to use `NPN_MemFlush` in situations where you cannot use `NPN_MemAlloc`, for example, when calling system methods that allocate memory indirectly. §

See Also

`NPN_MemFlush, NPN_MemFree`

NPN_MemFlush

Requests that the browser free a specified amount of memory.

Implemented only on Mac OS.

Syntax

```
#include <npapi.h>
uint32 NPN_MemFlush(uint32 size);
```

Parameters

The function has the following parameters:

`size`	Size of memory, in bytes, to free in the browser's memory space.

Returns

- If successful, the function returns the amount of freed memory, in bytes.
- If no memory can be freed, the plug-in returns 0.

Description

The plug-in calls `NPN_MemFlush` when it is not possible to call **NPN_MemAlloc**, for example, when calling system APIs that indirectly allocate memory. To request that the browser free as much memory as possible, call `NPN_MemFlush` repeatedly until it returns 0.

On Mac OS, you can use this method to free memory before calling memory-intensive Mac Toolbox calls.

In general, plug-ins should use `NPN_MemAlloc` to allocate memory in the browser's memory space, since this function automatically frees cached data if necessary to fulfill the request.

See Also

NPN_MemFlush, NPN_MemFree

NPN_MemFree

Deallocates a block of allocated memory.

Syntax

```
#include <npapi.h>
void NPN_MemFree (void* ptr);
```

Parameters

The function has the following parameters:

ptr	Block of memory previously allocated using NPN_MemAlloc.

Description

NPN_MemFree deallocates a block of memory that was allocated using **NPN_MemAlloc** only. NPN_MemFree does not free memory allocated by any other means.

See Also

NPN_MemAlloc, NPN_MemFlush

NPN_NewStream

Requests the creation of a new data stream produced by the plug-in and consumed by the browser.

Syntax

```
#include <npapi.h>
NPError NPN_NewStream(NPP         instance,
                      NPMIMEType  type,
                      const char* target,
                      NPStream**  stream);
```

Parameters

The function has the following parameters:

instance	Pointer to current plug-in instance.
type	MIME type of the stream.
target	Name of the target window or frame, or one of several special target names. For values, see **`NPN_GetURL`**.
stream	Stream to be created by the browser.

Returns

- If successful, the function returns `NPERR_NO_ERROR`.
- If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

`NPN_NewStream` creates a new stream of data produced by the plug-in and consumed by the browser.

The MIME parameter is the MIME type of the plug-in to create. A plug-in can create another instance of itself by specifying its own MIME type and a new target name in a call to `NPN_NewStream`.

The stream is returned in the stream parameter. The plug-in can use this object in subsequent calls to **NPN_Write** to write data into the stream. When the plug-in has written all of its data into the stream, **NPN_DestroyStream** terminates the stream and deallocates the `NPStream` object.

The target parameter is the name of the target window or frame, or one of several special target names. For parameter values and information about how to use them, see **NPN_GetURL**. If the new stream has the target of `_self`, this function should return an `INVALID_PARAM` `NPError`.

See Also

`NPP_NewStream`, **`NPP_Write`**, **`NPP_DestroyStream`**, **`NPStream`**, **`NPP`**

NPN_PostURL

Posts data to a URL.

Syntax

```
#include <npapi.h>
NPError NPN_PostURL(NPP instance, const char *url,
                    const char *target, uint32 len,
                    const char *buf, NPBool file);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the current plug-in instance.
`url`	URL of the request, specified by the plug-in.

`target`	Display target, specified by the plug-in. If null, pass the new stream back to the current plug-in instance regardless of MIME type. For values, see `NPN_GetURL`.
`len`	Length of the buffer buf.
`buf`	Path to local temporary file or data buffer that contains the data to post. Temporary file is deleted after use. Data in buffer cannot be posted for a protocol that requires a header.
`file`	A boolean value that specifies whether to post a file. Values: • true: Post the file whose the path is specified in buf, then delete the file. • false: Post the raw data in buf.

Returns

- If successful, the function returns `NPERR_NO_ERROR`.
- If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

`NPN_PostURL` works similarly to **NPN_GetURL**, but in reverse.

- `NPN_GetURL` reads data from the URL and either displays it in the target window or delivers it to the plug-in.
- `NPN_PostURL` writes data from a file or buffer to the URL and either displays the server's response in the target window or delivers it to the plug-in. If the target parameter is null, the new stream is passed to the plug-in regardless of MIME type.

When you use `NPN_PostURL` to send data to the server, you can handle the response in several different ways by specifying different target parameters.

- If target is null, the server response is sent back to the plug-in. You can get the data and save it in a file or use it in a program.
- If you specify `_current`, `_self`, or `_top`, the response data is written to the same plug-in window and the plug-in is unloaded.
- If you specify `_new` or `_blank`, the response data is written to a new browser window. You can also write the response data to a frame by specifying the frame name as the target parameter.

For HTTP URLs only, the browser resolves this method as the HTTP server method POST, which transmits data to the server.

The data to post can be contained either in a local temporary file or a new memory buffer.

- To post to a temporary file, set the flag file to true, the buffer buf to the path name string for a file, and len to the length of the path string. The file-type URL prefix "file://" is optional.

MS Windows and Mac OS

If a file is posted with any protocol other than FTP, the file must be text with Unix-style line breaks ('\n' separators only). §

- To post data from a memory buffer, set the flag file to false, the buffer buf to the data to post, and len to the length of buffer.

Possible URL types include HTTP (similar to an HTML form submission), mail (sending mail), news (posting a news article), and FTP (upload a file). Plug-ins can use this function to post form data to CGI scripts using HTTP or upload files to a remote server using FTP.

You cannot use `NPN_PostURL` to specify headers (even a blank line) in a memory buffer. To do this, use **NPN_PostURLNotify**.

For protocols in which the headers must be distinguished from the body, such as HTTP, the buffer or file should contain the headers, followed by a blank line, then the body. If no custom headers are required, simply add a blank line ('\n') to the beginning of the file or buffer.

`NPN_PostURL` is typically asynchronous: it returns immediately and only later handles the request. For this reason, you may find it useful to call `NPN_PostURLNotify` instead; this function notifies your plug-in upon successful or unsuccessful completion of the request.

See Also

`NPN_GetURL`, `NPN_GetURLNotify`, `NPN_PostURL`, `NPN_PostURLNotify`, `NPP`

NPN_PostURLNotify

Posts data to a URL, and receives notification of the result.

Syntax

```
#include <npapi.h>
NPError NPN_PostURLNotify(NPP         instance,
                          const char* url,
                          const char* target,
                          uint32      len,
                          const char* buf,
                          NPBool      file,
                          void*       notifyData);
```

Parameters

The function has the following parameters:

`instance`	Current plug-in instance, specified by the plug-in.
`url`	URL of the POST request, specified by the plug-in.
`target`	Target window, specified by the plug-in. For values, see `NPN_GetURL`.
`len`	Length of the buffer buf.
`buf`	Path to local temporary file or data buffer that contains the data to post.
`file`	Whether to post a file. Values: • `true`: Post the local file whose path is specified in buf, then delete the file. • `false`: Post the raw data in buf.
`notifydata`	Plug-in-private value for associating the request with the subsequent `NPP_URLNotify` call, which returns this value (see Description below).

Returns

- If successful, the function returns `NPERR_NO_ERROR`.
- If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

`NPN_PostURLNotify` functions identically to **NPN_PostURL**, with these exceptions:

- `NPN_PostURLNotify` supports specifying headers when posting a memory buffer.
- `NPN_PostURLNotify` calls `NPP_URLNotify` upon successful or unsuccessful completion of the request. For more information, see `NPN_PostURL`.

`NPN_PostURLNotify` is typically asynchronous: it returns immediately and only later handles the request and calls **NPP_URLNotify**.

If this function is called with a target parameter value of _self or a parent to `_self`, this function should return an `INVALID_PARAM` `NPError`. This is the only way to notify the plug-in once it is deleted. See **NPN_GetURL** for information about this parameter.

See Also

`NPN_GetURL`, `NPP_URLNotify`, `NPN_PostURL`

NPN_ReloadPlugins

Reloads all plug-ins in the Plugins directory.

Syntax

```
#include <npapi.h>
void NPN_ReloadPlugins(NPBool reloadPages);code
```

Parameters

The function has the following parameter:

`reloadPages`	Whether to reload pages. Values: • true: Reload pages. • false: Do not reload pages.

Description

`NPN_ReloadPlugins` reads the Plugins directory for the current platform and reinstalls all of the plug-ins it finds there.

Netscape Gecko knows about all installed plug-ins at start-up. If you add or remove any plug-ins, the browser does not see them until you restart it. `NPN_ReloadPlugins` allows you to install a new plug-in and load it, or to remove a plug-in, without having to restart the browser. You could use this function as part of the plug-in's installation process.

See Also

`NPN_Version`

NPN_RequestRead

Requests a range of bytes for a seekable stream.

Syntax

```
#include <npapi.h>
NPError NPN_RequestRead(NPStream*    stream,
                        NPByteRange* rangeList);
```

Parameters

The function has the following parameters:

`stream`	Stream of type `NP_SEEK` from which to read bytes. Communicator writes the requested bytes to the plug-in through subsequent calls to **NPP_WriteReady** and **NPP_Write**.
`rangeList`	Range of bytes in the form of a linked list of **NPByteRange** objects, each of which specifies a request for a range of bytes.

Returns

- If successful, the function returns `NPERR_NO_ERROR`.
- If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

For a seekable stream, the browser sends data only in response to requests by the plug-in. The plug-in calls `NPN_RequestRead` to request data from a seekable stream.

The plug-in can use this function to make one or more requests for ranges of bytes. These requests result in subsequent calls to **NPP_WriteReady** and **NPP_Write**. For multiple requests, the function creates a linked list of `NPByteRange` structures, each of which represents a separate request.

If the plug-in requests multiple ranges (either through a list of `NPByteRange` objects in a single call to `NPN_RequestRead`, or multiple calls to `NPN_RequestRead`), the browser can write individual ranges in any order, and with any number of **NPP_WriteReady** and **NPP_Write** calls.

The plug-in must allocate `NPByteRange` objects, which the browser copies if necessary. The plug-in can free these as soon as the call returns.

Seekable streams are created by calling **NPP_NewStream** with `NP_SEEK` as the stype mode.

- The plug-in can call `NPN_RequestRead` on streams that were not initially in `NP_SEEK` mode as long as the stream is inherently seekable; `NPN_RequestRead` automatically changes the mode to `NP_SEEK`.

- If the stream is not inherently seekable, the stream must have been put in `NP_SEEK` mode initially (since the browser must cache all the stream data on disk in order to access it randomly).
- If `NPN_RequestRead` is called on a stream that is not inherently seekable and not initially in mode `NP_SEEK`, it returns the error code `NPERR_STREAM_NOT_SEEKABLE`.

Typically, the only streams that are inherently seekable are those from in-memory or on-disk data, or from HTTP servers that support byte-range requests.

See Also

`NPP_NewStream`, `NPStream`

NPN_SetValue

Sets various modes of plug-in operation.

Syntax

```
#include <npapi.h>
NPError NPN_SetValue(NPP         instance,
                     NPPVariable variable,
                     void        *value);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the current plug-in instance.
`variable`	Values the function can set: • `NPPVpluginWindowBool`: Sets windowless mode for display of a plug-in; true=windowless, false=not windowless • `NPPVpluginTransparentBool`: Sets transparent mode for display of a plug-in; true=transparent, false=opaque • `NPPVjavascriptPushCallerBool` - Specifies whether you are pushing or popping the JSContext off the stack • `NPPVpluginKeepLibraryInMemory` - Tells browser that plugin dll should live longer than usual
`value`	The value of the specified variable to be set, `TRUE` or `FALSE`.

Returns

- If successful, the function returns `NPERR_NO_ERROR`.
- If unsuccessful, the plug-in is not loaded and the function returns an error code. For possible values, see **Error Codes**.

Description

A good place to set plug-in operation mode such as windowless mode is **NPP_New**, so the browser knows right away what mode the plug-in is designed to operate in.

`NPPVpluginWindowBool` (Windows and Unix) specifies that plug-in operates in windowless mode. In this mode no window messages are send to the plug-in as there is no window associated with it, all the browser to plug-in commucations related to drawing and mouse and keyboard input are event based and accomplished via **NPP_HandleEvent**. To set windowless operation plugin calls `NPN_SetValue` with `NPPVpluginWindowBool` as its variable parameter and `TRUE` as its value parameter. As a default, plug-ins are windowd, so if **NPP_New** does not contain this call the plug-in is considered to be windowed.

`NPPVpluginTransparentBool` (Windows and Unix) specifies that a plug-in is either opaque or transparent. To specify an opaque mode, the plugin calls `NPN_SetValue` with `NPPVpluginTransparentBool` for its variable parameter and `FALSE` for its value parameter. To specify a transparent mode, the value parameter should be set to `TRUE`.

`NPPVjavascriptPushCallerBool` sets whether you are pushing or popping the appropriate JSContext off of the stack (See the two-way scriptability article on the Mozilla Plug-ins project page for more details).

`NPPVpluginKeepLibraryInMemory` specifies that the plug-in does not want to be unloaded from memory after the page which initiated it has gone. Normally, when the browser navigates away from the page containing the plug-in all plug-in instances get `NPP_Destroy` call, and if there is no more instances of the plug-in active the plug-in is called its `NP_Shutdown` method and the plug-in dll gets unloaded from memory. If this is not desired the plug-in can instruct the browser not to unload the dll and not to call `NP_Shutdown` when the page is left. In such a case all this will be done on the browser shutdown. Plug-in calls `NPN_SetValue` any time with `NPPVpluginKeepLibraryInMemory` as variable parameter and value set to `TRUE`. By default, the dll will be unloaded from memory preceded by `NP_Shutdown` call.

Remarks

All four variable values are boolean. Although the function prototype has type of value void *, the actual boolean should be placed there, not a pointer to a boolean. The browser code reads this parameter as follows (`NPPVpluginWindowBool` as an example):

```
NPError NP_EXPORT _setvalue(NPP npp, NPPVariable
variable, void *value)
{
 ...
 BOOL bWindowless = (value == NULL);
 ...
}
```

So the proper way to call this function from a plug-in would be:

```
BOOL bWindowed = FALSE;
 NPN_SetValue(npp, NPPVpluginWindowBool, (
      void *)bWindowed);
```

See Also

NPP_New, NPN_GetValue, NPP_SetValue

NPN_Status

Displays a message on the status line of the browser window.

Syntax

```
#include <npapi.h>
void NPN_Status(NPP instance, const char* message);
```

Parameters

The function has the following parameters:

instance	Pointer to the current plug-in instance.
message	Pointer the buffer that contains the status message string to display.

Description

You can use this function to make your plug-in user interface simulate the browser ‘s behavior. When the user moves the cursor over a link in a browser window, Communicator displays information about it in the status message area (on the lower edge of the browser window). If your plug-in has a button or other object that acts as a link when clicked, you can call NPN_Status to display a description or URL when the user moves the cursor over it.

The browser always displays the last status line message it receives, regardless of the message source. Your message is always displayed, but you have no control over how long it stays in the status line before another message replaces it.

See Also

NPN_UserAgent, NPP

NPN_UserAgent

Returns the browser's user agent field.

Syntax

```
#include <npapi.h>
const char* NPN_UserAgent(NPP instance);
```

Parameters

The function has the following parameter:

instance	Pointer to the current plug-in instance.

Returns

A pointer to a buffer that contains the user agent field of the browser.

Description

The user agent is the part of the HTTP header that identifies the browser during transfers. You can use this information to verify that the expected browser is in use, or you can use it in combination with NPN_Version to supply different code for different versions of Netscape browsers.

See Also

`NPN_Status`, **`NPN_Version`**

NPN_Version

Returns version information for the Plug-in API.

Syntax

```
#include <npapi.h>
void NPN_Version(int*  plugin_major,
                 int*  plugin_minor,
                 int*  netscape_major,
                 int*  netscape_minor);
```

Parameters

The function has the following parameters:

`plugin_major`	Pointer to a plug-in's major version number; changes with major code release number.
`plugin_minor`	Pointer to a plug-in's minor version number; changes with point release number.
`netscape_major`	Pointer to the browser's major version; changes with major code release number.
`netscape_minor`	Pointer to the browser's version; changes with point release number.

Description

The values of the major and minor version numbers of the Plug-in API are determined when the plug-in and the browser are compiled. For example, Plug-in API version 4.03 has a major version number of 4 and a point release number of 3. This function gets the values from the plug-in rather than from the browser.

A plug-in can use this function to check that the version of the Plug-in API it is using is compatible with the version in use by the browser. This could be part of the initialization process. For more information and an example, see "Getting the Current Version."

You can use `NPN_Version` to inquire on version constants (`NPVERS` constants), which represent particular Communicator features. Once the plug-in obtains a version number, it can inquire on a version constant to find out if the feature it represents exists in this version. For example, the plug-in could inquire on the constant `NPVERS_HAS_WINDOWLESS` to see if it is running in a version of Communicator that

supports windowless functionality. For more information and an example, see "Finding Out if a Feature Exists." For a listing of version constants defined in the Plug-in API, see "Version Feature Constants."

NOTE: Platform-specific code in the Plug-in API files npwin.cpp, npmac.cpp, or npunix.c checks version numbers automatically. A plug-in whose major version is less than the major version of the browser is not loaded. §

See Also

`NPN_UserAgent`, **`NP_Initialize`**

NPN_Write

Pushes data into a stream produced by the plug-in and consumed by the browser.

Syntax

```
#include <npapi.h>
NPN_Write(NPP       instance,
          NPStream* stream,
          int32     len,
          void*      buf);
```

Parameters

The function has the following parameters:

`instance`	Pointer to the current plug-in instance.
`stream`	Pointer to the current stream.
`len`	Length in bytes of buf.
`buf`	Buffer of data delivered for the stream.

Returns

- If successful, the function returns a positive integer representing the number of bytes written (consumed by the browser). This number depends on the size of the browser's memory buffers, the number of active streams, and other factors.
- If unsuccessful, the plug-in returns a negative integer. This indicates that the browser encountered an error while processing the data, so the plug-in should terminate the stream by calling `NPN_DestroyStream`.

Description

`NPN_Write` delivers a buffer from the stream to the instance. A plug-in can call this function multiple times after creating a stream with `NPN_NewStream`. The browser makes a copy of the buffer if necessary, so the plug-in can free the buffer as the method returns, if desired. See "Example of Sending a Stream" for an example that includes `NPN_Write`.

See Also

`NPP_NewStream`, `NPP_DestroyStream`, `NPP_Write`, `NPStream`, `NPP`

Structures

This chapter describes the data structures that are used to represent the various objects in the plug-in API.

Structure Summary

NPAnyCallbackStruct	Contains information required during embedded mode printing.
NPByteRange	Represents a particular range of bytes from a stream.
NPEmbedPrint	Substructure of `NPPrint` that contains platform-specific information used during embedded mode printing.
NPEvent	Represents an event passed by `NPP_HandleEvent` to a windowless plug-in.
NPFullPrint	Substructure of `NPPrint` that contains platform-specific information used during full-page mode printing.
NPP	Represents a single instance of a plug-in.
NP_Port	Contains information required by the window field of an `NPWindow` structure.
NPPrint	Contains information the plug-in needs to print itself in full-page or embedded mode.
NPPrintCallbackStruct	Contains information required by the platformPrint field of the `NPEmbedPrint` during embedded mode printing.

NP_Rect	Represents a rectangular area of a page.
NP_Region	Represents a platform-defined region of a page.
NPSavedData	Block of instance information saved after the plug-in is deleted; can be returned to the plug-in.
NPSetWindowCallbackStruct	Contains information about the plug-in's Unix window environment.
NPStream	Represents a stream of data either produced by the browser and consumed by the plug-in, or produced by the plug-in and consumed by the browser.
NPWindow	Contains information about the target into which the plug-in instance can draw.

NPAnyCallbackStruct

Used on Unix only.

Contains information required during embedded mode printing.

Syntax

```
typedef struct
{
    int32 type;
} NPAnyCallbackStruct;
```

Fields

The data structure has the following field:

`type`	Always contains `NP_PRINT`.

Description

Callback structures are used to pass platform-specific information. The `NPAnyCallbackStruct` structure contains information required by the `platformPrint` field of the **NPEmbedPrint** structure during embedded mode printing.

During printing in embedded mode, the `platformPrint` field of the **NPEmbedPrint** structure points to an `NPAnyCallbackStruct`. This structure contains the file pointer to which the plug-in should write its Postscript data. At the time the plug-in is called, the browser has already opened the file and written Postscript for other parts of the page. When the plug-in is done, it should leave the file open, as the browser can continue to write additional Postscript data to the file.

See Also

`NPP_Print`, `NPEmbedPrint`, `NPSetWindowCallbackStruct`, `NPPrintCallbackStruct`

NPByteRange

Represents a particular range of bytes from a stream.

Syntax

```
typedef struct _NPByteRange
{
    int32  offset;  /* negative offset = from the end */
    uint32 length;
    struct _NPByteRange* next;
} NPByteRange;
```

Fields

The data structure has the following fields:

`offset`	Offset in bytes of the requested range, either positive or negative: • Positive value: Offset from the beginning of the stream. • Negative value: Offset from the end of the stream.
`length`	Number of bytes to fetch from the specified offset.
`next`	Points to the next `NPByteRange` request in the list of requests, or null if this is the last request.

Description

The plug-in seeks within a stream by building a linked list of one or more `NPByteRange` objects, which represents a set of discontiguous byte ranges. The only Plug-in API call that uses the `NPByteRange` type is **NPN_RequestRead**, which allows the plug-in to read specified parts of a file without downloading it.

The plug-in is responsible for deleting `NPByteRange` objects when finished with them. The browser makes a copy if it needs to keep the objects beyond the call to **NPN_RequestRead**.

See Also

`NPN_RequestRead`

NPEmbedPrint

Substructure of **NPPrint** that contains platform-specific information used during embedded mode printing.

Syntax

```
typedef struct _NPEmbedPrint
{
    NPWindow window;
    void*    platformPrint;  /* Platform-specific */
} NPEmbedPrint;
```

Fields

The data structure has the following fields:

`window`	The `NPWindow` the plug-in should use for printing.
`platformPrint`	Additional platform-specific printing information. • Mac OS: `THPrint` • Unix: Pointer to a `NPPrintCallbackStruct`.

Description

The **NPP_Print** function passes a pointer to an **NPPrint** object (previously allocated by the browser) to the plug-in. The `NPEmbedPrint` structure is used when the mode field of **NPPrint** is set to `NP_EMBED`.

Unix

The plug-in location and size in the `NPWindow` are in page coordinates (720/ inch), but the printer requires point coordinates (72/inch).

See Also

`NPFullPrint`, **`NP_Port`**, **`NPP_Print`**, **`NPPrint`**, **`NPPrintCallbackStruct`**

NPEvent

Represents an event passed by `NPP_HandleEvent` to a windowless plug-in.

Syntax

MS Windows

```
typedef struct _NPEvent
{
    uint16 event;
    uint32 wParam;
    uint32 lParam;
} NPEvent;
```

Mac OS

```
typedef EventRecord NPEvent;
TYPE EventRecord =
    RECORD {
       what:      Integer;
       message:   LongInt;
       when:      LongInt;
       where:     Point;
       modifiers: Integer;
    END;
```

XWindows

```
typedef XEvent NPEvent;
```

Fields

NPEvent on MS Windows

The data structure has the following fields:

`event`
One of the following event types:

- `WM_PAINT`
- `WM_LBUTTONDOWN`
- `WM_LBUTTONUP`
- `WM_LBUTTONDBLCLK`
- `WM_RBUTTONDOWN`
- `WM_RBUTTONUP`
- `WM_RBUTTONDBLCLK`
- `WM_MBUTTONDOWN`
- `WM_MBUTTONUP`
- `WM_MBUTTONDBLCLK`
- `WM_MOUSEMOVE`
- `WM_KEYUP`
- `WM_KEYDOWN`
- `WM_SETCURSOR`
- `WM_SETFOCUS`
- `WM_KILLFOCUS`

For information about these events, see your MS Windows documentation.

`wParam`
32 bit field for Windows event parameter; parameter value depends upon event type.

`lParam`
32 bit field for Windows event parameter; parameter value depends upon event type.

EventRecord NPEvent on Mac OS

`NPEvent` is defined as an EventRecord data structure, which has the following fields:

`what`	Integer representing an event type. Both windowed and windowless plug-ins receive the same events. Values:
	`0 nullEvent`
	`1 mouseDown`
	`2 mouseUp`
	`3 keyDown`
	`4 keyUp`
	`5 autoKey`
	`6 updateEvt`
	`7 diskEvt`
	`8 activateEvt`
	`15 osEvt`
	`23 kHighLevelEvent`
	`getFocusEvent 0, 1 (true, false)`
	`loseFocusEvent`
	`adjustCursorEvent 0, 1 (true, false)`
	
	For information about these events, see your Mac OS documentation.

`message`	`LongInt`. Additional information about the event. Type of information depends on the event type. Undefined for null, mouseUp, and mouseDown events.
`when`	`LongInt`. Ticks since start-up.
`where`	`Point`. Cursor location.
`modifiers`	`Integer`. Flags.

Description

MS Windows Description

The type `NPEvent` represents an event passed by **NPP_HandleEvent** to a windowless plug-in. For information about these events, see your MS Windows documentation.

Mac OS Description

The `NPEvent` object represents an event passed by **NPP_HandleEvent** to a windowless plug-in. This structure is defined as EventRecord, the event type used by Mac OS platform. On Mac OS, plug-ins receive the same events for both windowed and windowless plug-ins, as follows.

- Mouse events: Sent if the mouse is within the bounds of the instance.
- Key events: Sent if the instance has text focus (see below).
- Update events: Sent if the update region intersects the instance's bounds.
- Activate events: Sent to all instances in the window being activate or deactivated.
- Suspend/Resume events: Sent to all instances in all windows.
- Null events: Sent to all instances in all windows.

In addition to these standard types, the browser provides three additional event types that can be passed in the event->what field of the EventRecord:

- getFocusEvent: Sent when the instance could become the focus of subsequent key events, when the user clicks the instance or presses the tab key to focus the instance.
- If your instance accepts key events, return true, and key events will be sent to the instance until it receives a loseFocusEvent.
- If your plug-in ignores key events, return false, and the key events will be processed by Netscape itself.
- loseFocusEvent: Sent when the instance has lost the text focus, as a result of the user clicking elsewhere on the page or pressing the Tab key to move the focus. No key events are sent to the instance until the next `getFocusEvent`.
- adjustCursorEvent: Send when the mouse enters or leaves the bounds of the instance.
- If your plug-in wants to set the cursor when the mouse is within the instance, set the cursor and return true.
- If you don't want a special cursor, return false and the browser will use the standard arrow cursor.

XWindows Description

The `NPEvent` object represents an event passed by **NPP_HandleEvent** to a windowless plug-in. The `NPEvent` structure is defined as XEvent, the definition of the event type used by the XWindows platform. For information about the XEvent structure and XWindows events, see your XWindows documentation.

See Also

`NPP_HandleEvent`

NPFullPrint

Substructure of `NPPrint` that contains platform-specific information used during full-page mode printing.

Syntax

```
typedef struct _NPFullPrint
{
    NPBool pluginPrinted; /* true: print fullscreen */
    NPBool printOne;      /* true: print one copy */
                          /*       to default printer */
    void*  platformPrint; /* Platform-specific */
} NPFullPrint;
```

Fields

The data structure has the following fields:

`pluginPrinted`	Determines whether the plug-in prints in full-page mode. Values: • true: Plug-in takes complete control of the printing process and prints full-page. • false: (Default) Plug-in renders its area of the page only (for embedded plug-in).
`printOne`	Not currently in use. Should always be false. • true: Print single copy of page to the default printer. • false: Display print dialogs so user can choose printer, other options.
`platformPrint`	Platform-specific printing information. • Mac OS: `THPrint` • MS Windows: Printer's device context

Description

The **NPP_Print** function passes the plug-in a pointer to an **NPPrint** object (previously allocated by the browser). The `NPFullPrint` structure is used when the mode field of **NPPrint** is set to `NP_Full`.

The `pluginPrinted` field of this structure determines whether the plug-in prints in full-page mode or not. If you want the plug-in to take complete control of the printing process, it should print the full page and set the field pluginPrinted to true before returning.

If you want an embedded plug-in to simply render its area of the page, set pluginPrinted to false and return immediately; the browser calls **NPP_Print** again with the NPEmbedPrint substructure of NPPrint.

See Also

`NPP_Print`, `NPPrint`, `NPEmbedPrint`

NPP

Represents a single instance of a plug-in.

Syntax

```
typedef struct _NPP
{
    void* pdata; /* plug-in private data */
    void* ndata; /* Netscape private data */
} NPP_t;
typedef NPP_t*  NPP;
```

Fields

The data structure has the following fields:

`pdata`	Plug-in private value that a plug-in can use to store a pointer to an internal data structure associated with the instance; not modified by the browser.
`ndata`	Private browser value that can store data associated with the instance; should not be modified by the plug-in.

Description

Netscape Gecko creates an `NPP` structure for each plug-in instance and passes a pointer to it to **NPP_New**. This pointer identifies the instance on which API calls should operate and represents the opaque instance handle of a plug-in. `NPP` contains private instance data for both the plug-in and the browser.

The **NPP_Destroy** function informs the plug-in when the `NPP` instance is about to be deleted; after this call returns, the NPP pointer is no longer valid.

See Also

`NPP_New, NPP_Destroy`

NP_Port

Used on Mac OS only.

Contains information required by the window field of an NPWindow structure.

Syntax

```
typedef struct NP_Port
{
    CGrafPtr port;  /* Grafport */
    int32    portx; /* position inside the topmost
window */
    int32    porty;
} NP_Port;
```

Fields

The data structure has the following fields:

port	Standard Mac OS port into which the plug-in should draw.
portx, porty	Top-left corner of the plug-in rectangle in port coordinates (taking the scroll position into account).

Description

On Mac OS, the window field of an **NPWindow** structure points to an `NP_Port` object, which is allocated by the browser. The `NP_Port` is valid for the lifetime of the **NPWindow**, that is, until **NPP_SetWindow** is called again with a different value or the instance is destroyed.

Since the port is shared between the plug-in and other plug-ins and the browser, the plug-in should always do the following:

- Draw only within the area designated by the `NPWindow`.

- Save the current port settings before changing the port for drawing.
- Set the desired port settings before drawing.
- Restore the previous port settings after drawing.

See Also

NPP_SetWindow, **NPWindow**

NPPrint

Contains information the plug-in needs to print itself in full-page or embedded mode.

Syntax

```
typedef struct _NPPrint
{
    uint16 mode;   /* NP_FULL or NP_EMBED */
    union
    {
       NPFullPrint fullPrint;   /* if mode is NP_FULL */
        NPEmbedPrint embedPrint; /* if mode is NP_EMBED
*/
    } print;
} NPPrint;
```

Fields

The data structure has the following fields:

`mode` — Determines whether plug-in prints in full-page or embedded mode. Values:

- `NP_FULL`: Pointer to `NPFullPrint` structure. Plug-in can optionally print in full-page mode. The fullPrint field of the union is valid. See `NPFullPrint` and `NPP_Print`.
- `NP_EMBED`: Pointer to `NPEmbedPrint` structure. Plug-in should print in embedded mode. The embedPrint field of the union is valid. See `NPEmbedPrint`.

Description

The **NPP_Print** function passes a pointer to an `NPPrint` object (previously allocated by the browser) to the plug-in. The pointer and fields within the `NPPrint` structure are valid only for the duration of the **NPP_Print** call.

See Also

`NPP_Print`, `NPFullPrint`, `NPEmbedPrint`

NPPrintCallbackStruct

Used on Unix only.

Contains information required by the platformPrint field of the **NPEmbedPrint** during embedded mode printing.

Syntax

```
typedef struct
{
    int32  type;
    FILE*  fp;
} NPPrintCallbackStruct;
```

Fields

The data structure has the following fields:

`type`	Always contains `NP_PRINT`.
`fp`	Pointer to file to which the plug-in should write its Postscript data.

Description

Callback structures are used to pass platform-specific information. The `NPPrintCallbackStruct` structure contains the file pointer to which the plug-in should write its Postscript data. This information is required by the `platformPrint` field of the **NPEmbedPrint** structure during embedded mode printing.

At the time the plug-in is called, the browser has already opened the file and written Postscript for other parts of the page. When the plug-in is done, it should leave the file open, as the browser can continue to write additional Postscript data to the file.

See Also

`NPP_Print`, **`NPEmbedPrint`**, **`NPSetWindowCallbackStruct`**, **`NPAnyCallbackStruct`**

NP_Rect

Represents a rectangular area of a page.

Syntax

```
typedef struct _NPRect
{
    uint16 top;
    uint16 left;
    uint16 bottom;
    uint16 right;
} NPRect;
```

Fields

The data structure has the following fields:

`top, left, bottom, right`	Top, left side, bottom, and right side of the rectangle.

Description

`NPRect` defines the bounding box of the area of the plug-in window to be updated, painted, invalidated, or clipped to.

See Also

`NPN_ForceRedraw`, `NPN_InvalidateRect`, `NPN_InvalidateRegion`, `NP_Region`, `NPWindow`

NP_Region

Represents a platform-defined region of a page.

Syntax

MS Windows:

```
typedef HRGN NPRegion;
```

Mac OS:

```
typedef RgnHandle NPRegion;
```

XWindows:

```
typedef Region NPRegion;
```

Description

`NPRect` defines the region of the plug-in window to be updated, painted, invalidated, or clipped to. For information about the region type definition used by your platform, see your platform documentation.

See Also

`NPN_ForceRedraw`, **`NPN_InvalidateRect`**, **`NPN_InvalidateRegion`**, **`NP_Region`**, **`NPWindow`**

NPSavedData

Block of instance information saved after the plug-in is deleted; can be returned to the plug-in.

Syntax

```
typedef struct _NPSavedData
{
    int32  len;
    void*  buf;
} NPSavedData;
```

Fields

The data structure has the following fields:

`len`	Length in bytes of the buffer pointed to by buf; set by the plug-in.
`buf`	Pointer to a memory buffer allocated by the plug-in with `NPN_MemAlloc`. Can be any reasonable size; its contents are private to the plug-in and are not modified by the browser.

Description

The `NPSavedData` object contains a block of per-instance information that Communicator saves after the instance is deleted. This information can be returned to another instance of the same plug-in if the user returns to the web page that contains it.

You can use the plug-in's **NPP_Destroy** function to allocate an `NPSavedData` object using the **NPN_MemAlloc** function, fill in the fields, and return it to the browser as an output parameter. See "Instance Destruction" for a code example that shows how to use `NPSavedData`.

If the user revisits a web page that contains a plug-in, the browser returns the `NPSavedData` to the new instance of the plug-in in a call to **NPP_New**. After this, the plug-in is responsible for keeping or deleting the objects as necessary.

See Also

`NPP_New, NPP_Destroy`

NPSetWindowCallbackStruct

Used only on Unix.

Contains information about the plug-in's Unix window environment.

Syntax

```
typedef struct
{
    int32         type;
    Display*      display;
    Visual*       visual;
    Colormap      colormap;
    unsigned int  depth;
} NPSetWindowCallbackStruct;
```

Fields

The data structure has the following fields:

`type`	Always contains `NP_SetWindow`.
`display`	Standard X Toolkit attribute. Pointer to the Display structure that represents the browser-server connection.
`visual`	Standard X Toolkit attribute. X Visual used by the top-level shell window in the Netscape window hierarchy.
`colormap`	Standard X Toolkit attribute. Colormap for the plug-in window.
`depth`	tandard X Toolkit attribute. Depth of the plug-in window.

Description

Callback structures are used to pass platform-specific information. The `NPSetWindowCallbackStruct` object, allocated by the browser, contains information required for the `ws_info` field of an **NPWindow**.

The **NPP_SetWindow** function passes a pointer to this structure to the plug-in. The structure is valid for the lifetime of the **NPWindow**, that is, until **NPP_SetWindow** is called again or the instance is destroyed.

The type field of this structure always contains `NP_SetWindow`. The remaining fields are Standard X Toolkit attributes of the top-level shell window in the browser window hierarchy.

See Also

NPP_SetWindow, **NPWindow**, **NPPrintCallbackStruct**, **NPAnyCallbackStruct**

NPStream

Represents a stream of data either produced by the browser and consumed by the plug-in, or produced by the plug-in and consumed by the browser.

Syntax

```
typedef struct _NPStream

{
    void* pdata;    /* plug-in private data */
    void* ndata;    /* Netscape private data */
    const char* url;
    uint32      end;
    uint32      lastmodified;
    void*       notifyData;
} NPStream;
```

Fields

The data structure has the following fields: Plug-in-private value that the plug-in can use to store a pointer to private data associated with the instance; not modified by the browser.

`ndata`	Browser-private value that can store data associated with the instance; should not be modified by the plug-in.
`url`	The URL that the data in the stream is read from or written to.
`end`	Offset in bytes of the end of the stream (equivalent to the length of the stream in bytes). Can be zero for streams of unknown length, such as streams returned from older FTP servers or generated "on the fly" by CGI scripts.
`lastmodified`	Time the data in the URL was last modified (if applicable), measured in seconds since 12:00 midnight GMT, January 1, 1970.
`notifyData`	Used only for streams generated in response to a `NPN_GetURLNotify` or `NPN_PostURLNotify` request. • For these streams, `notifyData` is set to the value of the notifyData parameter to `NPN_GetURLNotify` or `NPN_PostURLNotify`. • For other streams, notifyData is null.

Description

The browser allocates and initializes the `NPStream` object and passes it to the plug-in in as a parameter to **NPP_NewStream** or **NPN_NewStream**. The browser cannot delete the object until after it calls **NPP_DestroyStream** or the plug-in calls **NPN_DestroyStream**.

Streams produced by the browser: the browser creates the `NPStream` object and passes it to the plug-in initially as a parameter to `NPP_NewStream`. All API calls that operate on the stream (such as `NPP_WriteReady` and `NPP_Write`) use a pointer to this stream. The browser informs the plug-in when the stream is about to be deleted through `NPP_DestroyStream`, after which the `NPStream` object is no longer valid.

Streams produced by the plug-in: the browser creates the `NPStream` object and returns it as an output parameter when the plug-in calls `NPP_NewStream`. The plug-in must pass a pointer to the `NPStream` to all API calls that operate on the stream, such as `NPN_Write` and `NPN_DestroyStream`.

See Also

`NPP_NewStream`, **`NPP_DestroyStream`**, **`NPP_DestroyStream`**

NPWindow

Contains information about the target into which the plug-in instance can draw.

Syntax

```
typedef struct _NPWindow
{
    void*     window;   /* Platform specific handle */
    uint32    x;        /* Coordinates of top left corner */
    uint32    y;        /*   relative to a Netscape page */
    uint32    width;    /* Maximum window size */
    uint32    height;
    NPRect    clipRect; /* Clipping rectangle coordinates */
                        /*   in port - Used by Mac only */
#ifdef XP_UNIX
    void *    ws_info;  /* Platform-dependent additional data */
#endif /* XP_UNIX */
    NPWindowType type;   /* Window or drawable target */
} NPWindow;
```

Fields

The data structure has the following fields:

`window`	Platform-specific handle to a native window element in the Netscape window hierarchy on Windows (`HWND`) and Unix (X Window ID). Mac OS: window is a pointer to an `NP_Port`.
`x, y`	The x and y coordinates for the top left corner of the plug-in relative to the page (and thus relative to the origin of the drawable). Should not be modified by the plug-in.

`height, width`	The height and width of the plug-in area. Should not be modified by the plug-in.
`clipRect`	Clipping rectangle of the plug-in; the origin is the top left corner of the drawable or window. Clipping to the clipRect prevents the plug-in from overwriting the status bar, scroll bars, and other page elements when partially scrolled off the screen. Mac OS: clipRect is the rectangle in port coordinates to which the plug-in should clip its drawing.
`ws_info`	Unix: Contains information about the plug-in's Unix window environment; points to an `NPSetWindowCallbackStruct`.

`type`	`NPWindowType` value that specifies whether the `NPWindow` instance represents a window or a drawable. Values:
	• `NPWindowTypeWindow`: Indicates that the window field holds a platform-specific handle to a window (as in Navigator 2.0 and Navigator 3.0). The plug-in is considered windowed.
	• `NPWindowTypeDrawable`: Indicates that the window field holds a platform-specific handle to a drawable or an off-screen pixmap. The plug-in is considered windowless. Values:
	  • Windows: HDC
	  • Mac OS: pointer to `NP_Port` structure

Description

The `NPWindow` structure represents the native window or a drawable, and contains information about coordinate position, size, whether the plug-in is windowed or windowless, and some platform-specific information. The plug-in area is a native window element on Windows and Unix, or a rectangle within a native window on Mac OS. The x, y, height, and width coordinates of `NPWindow` specify the position and size of this area.

The browser calls **NPP_SetValue** whenever the drawable changes.

A windowed plug-in is drawn into a native window (or portion of a native window) on a web page. For windowed plug-ins, the browser calls the **NPP_SetWindow** method with an **NPWindow** structure that represents a drawable (a pointer to an `NPWindow` allocated by the browser). This window is valid until **NPP_SetWindow** is called again with a different window or the instance is destroyed.

A windowless plug-in is drawn into a target called a drawable, which can be defined in several ways depending on the platform. For windowless plug-ins, the browser calls the **NPP_SetWindow** method with an **NPWindow** structure that represents a drawable.

The plug-in should not modify the field values in this structure.

See Also

`NPP_SetWindow`, **`NP_Port`**, **`NPSetWindowCallbackStruct`**, **`NP_Rect`**

Constants

This section is a reference to the program definitions used by the Plug-in API. All program definitions are found in npapi.h.

- **Error Codes**
- **Result Codes**
- **Plug-in Version Constants**
- **Version Feature Constants**

Error Codes

Code	Value	Description
NPERR_NO_ERROR	0	No errors occurred.
NPERR_GENERIC_ERROR	1	Error with no specific error code occurred.
NPERR_INVALID_INSTANCE_ERROR	2	Invalid instance passed to the plug-in.
NPERR_INVALID_FUNCTABLE_ERROR	3	Function table invalid.
NPERR_MODULE_LOAD_FAILED_ERROR	4	Loading of plug-in failed.
NPERR_OUT_OF_MEMORY_ERROR	5	Memory allocation failed.
NPERR_INVALID_PLUGIN_ERROR	6	Plug-in missing or invalid.
NPERR_INVALID_PLUGIN_DIR_ERROR	7	Plug-in directory missing or invalid.
NPERR_INCOMPATIBLE_VERSION_ERROR	8	Versions of plug-in and Communicator do not match.
NPERR_INVALID_PARAM	9	Parameter missing or invalid.

NPERR_INVALID_URL	10	URL missing or invalid.
NPERR_FILE_NOT_FOUND	11	File missing or invalid.
NPERR_NO_DATA	12	Stream contains no data.
NPERR_STREAM_NOT_SEEKABLE	13	Seekable stream expected.

Result Codes

Constant	Value	Description
NPRES_DONE	0	(Most common): Completed normally; all data was sent to the instance.
NPRES_NETWORK_ERR	1	Stream failed due to problems with network, disk I/O, lack of memory, or other problems.
NPRES_USER_BREAK	2	User canceled stream directly by clicking the Stop button or indirectly by some action such as deleting the instance or initiating higher-priority network operations.

Plug-in Version Constants

Constant	Value	Description
NP_VERSION_MAJOR	0	Major version number; changes with major code release number.
NP_VERSION_MINOR	11	Minor version number; changes with point release number.

Version Feature Constants

NPVERS Constant: Version Feature Information	Value	Supported Feature
NPVERS_HAS_STREAMOUTPUT	8	Streaming data.
NPVERS_HAS_NOTIFICATION	9	Notification of completion.
NPVERS_HAS_LIVECONNECT	9	LiveConnect.
NPVERS_WIN16_HAS_LIVECONNECT	9	LiveConnect (Win16).
NPVERS_68K_HAS_LIVECONNECT	11	LiveConnect (68K).
NPVERS_HAS_WINDOWLESS	11	Windowless plug-in.
NPVERS_HAS_XPCONNECT_SCRIPTING	13	Scriptable plug-in.