

# Eidos Patch System

## Client-Side Reference Manual for Playstation®2 and Microsoft Windows

*Version 1.2*

*Updated May 18, 2005*

Prepared by Pinniped Software

Pinniped Software



game and internet programming

Copyright © 2005, Eidos, Inc. All rights reserved.



## Table of Contents

Overview.....	1
Patcher Class.....	2
Usage.....	2
Methods.....	3
Create.....	3
Destroy.....	3
GetPatcher.....	4
Update.....	4
GetPatchError.....	4
BeginVersionCheck.....	5
DownloadDescription.....	6
DownloadPatch.....	6
GetBytesDownloaded.....	7
PreparePatch.....	8
SetPatch.....	8
TestCompletion.....	9
WaitCompletion.....	10
PatchExtractor Class.....	11
Usage.....	11
Methods.....	11
Constructors.....	11
Init.....	12
IsGood.....	12
QueryVersion.....	13
SetSourceBuffer.....	13
SetSourceFile.....	13
SetSourceReadFunction.....	14
PatchExists.....	15
PatchItem.....	15
Methods Only on Playstation®2.....	16
LoadModule.....	16
FilePatcher Class.....	18
Methods.....	18
Constructors.....	18
Init.....	18
FindNextFile.....	19
PatchCurrentFile.....	19
Global Functions.....	21
Functions only on Microsoft Windows.....	21
LaunchPatcherApp.....	21
Error Codes.....	22



# Eidos Patch System Client-Side Reference Manual

*Version 1.2*

## Overview

---

This manual contains detailed information on the classes and methods used patch your game on both the Playstation®2 and the Windows PC. For general information regarding how to use the patch system and for platform-specific information, see the *Client-Side Programming Guide* for your hardware.

The following classes are described below:

- **Patcher** – Discover, download, and decrypt a patch from an HTTP server.
- **PatchExtractor** – Extract individual files from a composite patch, bringing the existing version of the file up to the patched revision.
- **FilePatcher** – (Windows only) Higher level interface over the **PatchExtractor** which will patch files directly on disk.

## Patcher Class

---

The **Patcher** class is used to acquire and prepare a patch for application. It handles all network communication, encryption (for memory card storage on the Playstation®2), and decryption.

### Usage

Include:

```
#include <PatchLib/Patcher.h>
```

Libraries:

```
PatchLib.a for Playstation®2, PatchLib.lib for PC.
```

- The **Patcher** class is meant to be a singleton; that is, only one instance of it is allowed. Thus, it cannot be created except through its `Create` method. Likewise, when you are done with it, you can call the `Destroy` method to make sure any memory associated with the **Patcher** or any resources (sockets, etc.) have been freed. Once the **Patcher** singleton has been destroyed, the **Patcher** code can be unloaded if so desired.
- Because the **Patcher** does much of its work in the background, your game is required to call the `Update` method frequently (such as once per frame) as long as the **Patcher** is instantiated.
- Some methods start an asynchronous task, and thus their completion merely indicates the task was started. You must use the `TestCompletion` method to discover when these tasks are complete and to get their status.
- The **Patcher** class can download and decrypt any number of patches. For each patch desired, the sequence is as follows:
  1. To get a pointer to the **Patcher** singleton at any time, use the `GetPatcher` method.
  2. Use `BeginVersionCheck` to start a test for patch availability.
  3. When the version check is done, `TestCompletion` will indicate whether a patch is needed.
  4. When a patch is needed, optionally use `DownloadDescription` to get a description of the patch to display to the user, and allow the user to select optional patches.

### Windows Only:

5. Use the `LaunchPatcherApp` function to launch the patching application.

### Playstation®2 Only:

5. When a patch is needed or selected, `DownloadPatch` will start the download of the patch itself.
6. When the download is done, `TestCompletion` will indicate whether it was downloaded successfully or not. The patch will be personalized with DNAS and can be stored on the memory card at this point.

7. After a patch has been downloaded (and stored on the memory card for Playstation®2), it must be decrypted with `PreparePatch`. Once `TestCompletion` indicates it is finished, the patch is ready to be applied with a **PatchExtractor**.

## Methods

---

### Create

```
static Patcher::Status Create(int threadPriority)
```

---

#### Description

One-time initialization function to be called at engine startup time or when the game determines that it may need to check for patches. Creates the static singleton **Patcher** instance. Note: `Create` MUST be called before any other **Patcher** functions.

#### Parameters

- `int threadPriority` – For Playstation®2, this priority determines the priority assigned to the background thread used by the **Patcher**. Make sure the priority is high enough that the thread will run. It does no harm (and in fact may be preferable) to have the priority higher than the main thread. This parameter is ignored on Windows.

#### Return Values

- `Patcher::PATCH_STATUS_OK` – The **Patcher** was created successfully.
- `Patcher::PATCH_STATUS_BAD_REQUEST` – The **Patcher** already exists. This return value is informational; no harm is done by attempting to instantiate the **Patcher** more than once.

---

### Destroy

```
static void Destroy(void)
```

---

#### Description

`Destroy` frees all allocated resources in the singleton **Patcher**, and indicates that no further **Patcher** calls will be made. The **Patcher** DLL can be removed from memory at this point.

#### Parameters

None.

#### Return Value

None.

---

## GetPatcher

`static Patcher* GetPatcher(void)`

---

### Description

Get a pointer to the singleton **Patcher**. The result is undefined if `Create` has not been called (don't count on a NULL return). Use this function to get a **Patcher** pointer for calling non-static methods.

### Parameters

None.

### Return Value

A pointer to the singleton **Patcher**. If the patcher has not been created, the return value is undefined.

---

## Update

`void Update(float dt)`

---

### Description

After the **Patcher** has been created, the `Update` function should be called frequently, such as once per frame, to provide a synchronization point with the game engine and to check for completion of asynchronous operations.

### Parameters

- `float dt` – The time since the last call to `Update`. This parameter is ignored in the current version, but may be used for handling timeouts in future versions.

### Return Values

None.

---

## GetPatchError

`PatchError GetPatchError(void)`

---

### Description

When any **Patcher** method indicates an error, `GetPatchError` is used to retrieve a more detailed error code. The possible return values are detailed under *Return Values* below.

### Parameters

None.

### Return Values

See the *Error Code* section at the end of this document for the possible error codes and their meaning.



---

## BeginVersionCheck

```
Patcher::Status BeginVersionCheck(const char* host, const char* path,  
                                   unsigned int version)
```

---

### Description

`BeginVersionCheck` will begin the check to see whether a patch is available or not, and collect information (such as size, etc.) about the patch. This function begins an asynchronous operation, so use `TestCompletion` to determine when the check is finished.

If the given host has the version description file given by `path`, the version number contained in that file is compared with the version number given with `version`. If the numbers do not match, `TestCompletion` will indicate an update is needed.

If the given host does not have the requested version description file, it is assumed no update is available; an error is *not* indicated.

### Parameters

- `const char* host` – Name of host holding the version description file. For example, *mygame.patch.eidos.com*.
- `const char* path` – Full path and file name of the patch description file. The path must be complete. For example, */assets/version.txt*.
- `unsigned int version` – The version number of the currently running application (or portion thereof if the patch is divided into parts). If this version number does not match that found in the version description file, the application will be told a patch is required.

### Return Values

- `Patcher::PATCH_STATUS_OK` – The version check was successfully initiated. Use `TestCompletion` to find out when the check is finished.
- `Patcher::PATCH_STATUS_INIT_FAILED` – This error is serious, and should never occur. It indicates that the **Patcher** was unable to perform a basic operation such as creating the background thread or a synchronization object. The most likely cause is that the system is out of memory.
- `Patcher::PATCH_STATUS_BAD_REQUEST` – Indicates that either a parameter (`host` or `path`) was bad, or that the **Patcher** is not in a state where a version check can be initiated (for example, it is busy performing another operation). If your game gets this status, check the following items:
  - `host` is a valid string (not null) and is less than 64 characters long,
  - `path` is a valid string (not null) and is less than 128 characters long,
  - you are properly calling `Update` each frame, and
  - you are not calling this function (or other asynchronous functions) multiple times in a row without `TestCompletion` indicating it is finished.

---

## DownloadDescription

`Patcher::Status DownloadDescription(char* descBuf, int bufSize)`

---

### Description

Once a patch is determined to be available, `DownloadDescription` will request the description file for the patch (normally a text description) from the download server. The file will be returned in `descBuf`, which must be at least `bufSize` bytes long. If the file is longer than `bufSize`, it is truncated to fit in the buffer.

`DownloadDescription` always makes sure your description has at least one null byte at the end, as a safeguard against buffer overflow attacks and to allow string processing on the buffer. Consequently, when the file is truncated to fit in the buffer, only `bufSize-1` bytes of the file will be returned to allow the null byte at the end.

Use `TestCompletion` to determine when the file has been completely downloaded.

### Parameters

- `char* descBuf` – The buffer to receive the description file.
- `int bufSize` – The size of `descBuf` in bytes.

### Return Values

- `Patcher::PATCH_STATUS_OK` – A description is available and the download has been initiated. Use `TestCompletion` to wait for the download to be complete.
- `Patcher::PATCH_STATUS_NOT_FOUND` – When performing the version check, no patch description file was described, and thus no description is available.
- `Patcher::PATCH_STATUS_BAD_REQUEST` – This status indicates a program error. Either a parameter was bad, or a patch was never determined to be available. If your game gets this status, check the following:
  - `descBuf` is a valid (non-null) pointer,
  - `bufSize` is greater than zero,
  - there are no other outstanding asynchronous requests (such as `BeginVersionCheck` or `DownloadPatch`), and
  - `TestCompletion` previously indicated there was a patch available by providing a positive value for `reqBufSize`.

---

## DownloadPatch

`Patcher::Status DownloadPatch(char* buffer, int bufferSize)`

---

### Description

`DownloadPatch` begins the process of downloading and (when required) encrypting a patch. This function should only be called after a version check (see `BeginVersionCheck`) has indicated a patch is

available. Use `TestCompletion` to determine when the download is complete and to get the status of the download.

**Note:** on Playstation®2, the patch in memory after a download is complete has been encrypted, or *personalized*, with DNAS, so it is ready to be stored on a memory card.

After the download is complete, use the `PreparePatch` method to prepare the patch before it can be applied.

### Parameters

- `char* buffer` – A buffer to receive the download. The buffer must be large enough to hold the entire download and each of its encrypted and decrypted states. The value can be obtained from the `reqBufSize` field of the `Patcher::CompletionInfo` structure. See the description of `TestCompletion` for more information.
- `int bufferSize` – The size of buffer, in bytes.

### Return Values

- `Patcher::PATCH_STATUS_OK` – The download was initiated successfully. Use `TestCompletion` to determine when the download is complete and to get the status of the download.
- `Patcher::PATCH_STATUS_BAD_REQUEST` – This status indicates a program error. Either a parameter was bad, or a patch was never determined to be available. If your game gets this status, check the following:
  - `buffer` is a valid (non-null) pointer,
  - `bufferSize` is greater than zero,
  - there are no other outstanding asynchronous requests (such as `BeginVersionCheck` or another call to `DownloadPatch`), and
  - `TestCompletion` previously indicated there was a patch available by providing a positive value for `reqBufSize`.

---

## GetBytesDownloaded

```
int GetBytesDownloaded(void)
```

---

### Description

After a download has been initiated with `DownloadPatch`, this function returns the total number of bytes downloaded so far. Thus detailed progress information can be displayed for the user during large downloads.

### Parameters

None.

### Return Value

Returns the number of bytes downloaded since `DownloadPatch` was called. If a patch is not being downloaded, the value is undefined.

---

## PreparePatch

`Patcher::Status PreparePatch(void)`

---

### Description

Begin to prepare the current patch for use. A patch must have been completely downloaded (or, for Playstation®2, read from the memory card) prior to calling this function. Use `TestCompletion` to wait for the preparation to complete and to get the final status.

The actual action of preparing the patch is system-dependent. For example, on the Playstation®2, preparation involves decrypting the personalized patch.

### Parameters

None.

### Return Values

- `Patcher::PATCH_STATUS_OK` – The preparation was initiated successfully.
- `Patcher::PATCH_STATUS_BAD_REQUEST` – The **Patcher** is not in a state ready to prepare the patch. If this error occurs, make sure you are properly checking `TestCompletion` for each asynchronous method you call, and that a patch was successfully downloaded.

---

## SetPatch

`Patcher::Status SetPatch(char* buffer, int bufferSize)`

---

### Description

**Note:** This function is intended for use with the Playstation®2. It is of little to no value on other platforms, although its use is not prohibited.

`SetPatch` takes a patch in its downloaded or stored state (e.g., an encrypted patch read from the memory card), and hands it to the **Patcher**. Your game can then proceed as if a patch were just successfully downloaded, by decrypting the patch with `PreparePatch`.

**Important:** the buffer is not copied, so you must make sure the pointer passed in for the buffer remains valid for the duration that this patch will be referenced (until the **Patcher** is destroyed or another patch is set or downloaded).

### Parameters

- `char* buffer` – The buffer containing the encrypted patch.
- `int bufferSize` – The size of the patch data in the buffer.

## Return Values

- `Patcher::PATCH_STATUS_OK` – The patch was successfully handed to the Patcher. Note that a good status here does not indicate the patch has been determined to be good, just that the Patcher has been made aware of this buffer.
- `Patcher::PATCH_STATUS_INIT_FAILED` – This error is serious, and should never occur. It indicates that the **Patcher** was unable to perform a basic operation such as creating the background thread or a synchronization object. The most likely cause is that the system is out of memory.
- `Patcher::PATCH_STATUS_BAD_REQUEST` – The patcher is not in a ready state. Make sure that there are no asynchronous operations pending.

---

## TestCompletion

`Patcher::Status TestCompletion(Patcher::CompletionInfo& info)`

---

### Description

`TestCompletion` is used whenever you have an asynchronous operation pending, as initiated by `BeginVersionCheck`, `DownloadPatch`, or `PreparePatch`. The return status indicates the status of the pending operation, and the `info` parameter will return size information related to the current operation when it completes.

In general, when `TestCompletion` returns `Patcher::PATCH_STATUS_OK`, the operation is complete and `info` contains further information. When `TestCompletion` returns `Patcher::PATCH_STATUS_BUSY`, the operation is still in progress.

To get size information back from `TestCompletion`, you must provide a `CompletionInfo` structure to be filled out if the operation is complete:

```
Patcher::CompletionInfo info;
Patcher* pPatcher = Patcher::GetPatcher();
Patcher::Status status = pPatcher->TestCompletion(info);
switch (status)
    [...]
```

If `TestCompletion` returns `Patcher::PATCH_STATUS_OK`, the fields in `info` will be filled out. The fields are as follows:

- `info.reqBufSize` – Upon completion of a version check initiated with `BeginVersionCheck`, `reqBufSize` will contain the size of the buffer required to download the patch from the server. It should be the largest of the three sizes provided.
- `info.storeSize` – The `storeSize` is the size required to store the patch on the memory card for Playstation®2. For Windows, this size is not needed. Note that upon completion of a version check (`BeginVersionCheck`), this size may be an estimate only. When a `DownloadPatch` operation is complete, the `storeSize` value will be accurate, because it is recomputed during the encryption (personalization) process.
- `info.clearSize` – The `clearSize` is the size of the final decrypted patch. Similar to `storeSize`, it may represent an estimate after completion of `BeginVersionCheck` or

`DownloadPatch`, but its value will be accurate after completion of `PreparePatch`. Make sure to save this value after preparing a patch for use in extracting data from the patch, because if it is wrong the checksums will indicate a failure at that point.

- `info.mandatory` – The `mandatory` field is a `bool` indicating whether this patch should be considered mandatory. The exact interpretation of the parameter is left up to the game, but it is generally used to differentiate mandatory patches that must be applied before joining an on-line game from optional asset updates. If `mandatory` is `true`, the patch is considered mandatory.

### Parameters

- `Patcher::CompletionInfo& info` – The information to be filled out if the operation is complete. If the operation is incomplete or an error occurred, the fields in `info` are undefined.

### Return Values

- `Patcher::PATCH_STATUS_OK` – The operation being tested is complete, and `info` contains relevant information.
- `Patcher::PATCH_STATUS_BUSY` – The asynchronous operation being tested is not yet complete. The `info` structure is not filled in.
- `Patcher::PATCH_STATUS_NET_ERROR` – The operation being tested is complete, but it encountered an error. The `info` structure is not filled in.

---

## WaitCompletion

`Patcher::Status` **WaitCompletion** (`Patcher::CompletionInfo& info`)

---

`WaitCompletion` is the same as `TestCompletion`, except that if the operation being tested is not complete, it waits until it is complete. Most games which access the **Patcher** class from the main thread will not use this function, but it is useful in the case that the patching code is being run from a thread that is not critical (not drawing or testing user input), such as some multi-threaded Windows games.

See `TestCompletion` for full details of operation.

## PatchExtractor Class

---

The **PatchExtractor** class is a low level interface that allows for “extraction” of files from a patch buffer previously downloaded and prepared with the **Patcher** class. It is primarily intended for use on the Playstation®2; Windows PC games will normally use the external patching application (see `LaunchPatcherApp`).

### Usage

Include:

```
#include <PatchExt/PatchExtractor.h>
```

Libraries:

```
PatchExt.a for Playstation®2, PatchExt.lib for PC.
```

The **PatchExtractor** class takes a clear-text buffer from the caller which has previously been prepared using `Patcher::PreparePatch`. The buffer can be handed to the `PatchExtractor` either at construction time or using the `Init` function. (A **PatchExtractor** can be constructed with no parameters so that it can be included as a member in one of your own classes.) So the code

```
PatchExtractor ext(buf, bufsize);
if (ext.IsGood()) ...
```

and

```
PatchExtractor ext;
if (ext.Init(buf, bufsize)) ...
```

are equivalent.

Once a **PatchExtractor** has been given a valid buffer and buffer size (which **MUST** be accurate; see the `Init` method below), you should check that it passes the integrity checks with the `IsGood` method. Assuming the integrity checks are good, you are ready to extract patched files using the **PatchExtractor**.

For full details on how to access your original data using read functions and other high level aspects of the **PatchExtractor**, see the *Eidos Patch System Client-Side Programming Guide for Playstation®2*.

### Methods

---

#### Constructors

```
PatchExtractor()
PatchExtractor(char* patchData, int dataSize)
```

---

#### Description

Two constructors are available for the **PatchExtractor**. The default constructor just sets up an empty extractor; you will later need to use the `Init` method to hand it a buffer of patch data.

The constructor which takes the patch data information simply calls the `Init` method with those

parameters. See the documentation of the `Init` method for details.

---

## Init

```
bool Init(char* patchData, int dataSize)
```

---

### Description

The `Init` method hands the given patch data to the **PatchExtractor** and verifies its integrity. The return value indicates whether the patch has passed the integrity checks.

**Note:** the `dataSize` parameter must be accurate in order to calculate a proper signature for the data. Make sure that it matches the actual data size (as determined by `clearSize` after preparing the data with `Patcher::PreparePatch`), and not some rounded-up size.

### Parameters

- `char* patchData` – The address of the buffer containing the patch data as prepared by the **Patcher**.
- `int dataSize` – The size of the patch data in the buffer. This size must match the size of the data in the buffer, not the allocated size of the buffer (see note above).

### Return Values

- `true` – The patch data passed the integrity tests and is ready for patching. A subsequent call to `IsGood` is guaranteed to be true also.
- `false` – The patch data failed the integrity test. If the integrity check fails when you think it should succeed, make sure the `dataSize` parameter is the correct size, and not a buffer size (see note above). The `dataSize` parameter must have exactly the right value for the integrity check to pass.

---

## IsGood

```
bool IsGood(void)
```

---

### Description

The `IsGood` function is used to test whether the integrity test of the patch succeeded when constructing a **PatchExtractor**.

### Parameters

None.

### Return Values

- `true` – The patch data passed the integrity tests and is ready for patching.
- `false` – The patch data failed the integrity test. If the integrity check fails when you think it should succeed, make sure the `dataSize` parameter is the correct size, and not a buffer size (see note above). The `dataSize` parameter must have exactly the right value for the integrity check to pass.



---

## QueryVersion

```
int QueryVersion(void)
```

---

### Description

`QueryVersion` returns the current version number as specified by the prepared patch currently in memory. Returns zero on error or if there is no patch available.

For Playstation®2, use `QueryVersion` on a patch loaded from the memory card in order to get your current version number for use with `Patcher::BeginVersionCheck`.

### Parameters

None.

### Return Value

Returns the version number stored in the current patch buffer. If the buffer is not in a good state (as indicated by the earlier call to `IsGood` or `Init`), zero is returned.

---

## SetSourceBuffer

```
void SetSourceBuffer(void* ptr)
```

---

### Description

As described in the *Eidos Patch System Programming Guide for Playstation®2*, a patch is applied to an individual file (or resource) in the patch buffer by modifying the original data. Thus before applying a patch to a file a method should be set up for accessing the original unpatched data.

`SetSourceBuffer` tells the **PatchExtractor** that the entire original data file resides in the buffer pointed to by `ptr`. Call this method before patching any individual item.

### Parameters

- `void* ptr` – A pointer to the original unpatched data file in memory. This buffer will be used to patch the next item requested.

### Return Value

None.

---

## SetSourceFile

```
void SetSourceFile(void)
```

---

### Description

As described in the *Eidos Patch System Programming Guide for Playstation®2*, a patch is applied to an individual file (or resource) in the patch buffer by modifying the original data. Thus before applying a patch to a file a method should be set up for accessing the original unpatched data.

The `SetSourceFile` method tells the **PatchExtractor** that the original version of the next file to be patched should be read from disk (from the DVD on Playstation®2). This method is the default method and is useful for rapid prototyping, but should rarely be used because of performance considerations.

---

## SetSourceReadFunction

```
void SetSourceReadFunction(PatcherReadFn* ptr, void* handle)
```

---

### Description

As described in the *Eidos Patch System Programming Guide for Playstation®2*, a patch is applied to an individual file (or resource) in the patch buffer by modifying the original data. Thus before applying a patch to a file a method should be set up for accessing the original unpatched data.

The `SetSourceReadFunction` method tells the **PatchExtractor** to call the given user-supplied function to access the original data for the next file to be patched. The read function has the following prototype:

```
int MyPatcherReadFn(void* handle, char* buf, int offset, int nbytes);
```

The read function parameters are as follows:

- `void* handle` – This parameter is the original handle you passed to `SetSourceReadFunction`. You can use it to store any information you need to access your data.
- `char* buf` – The buffer to be filled. Your function should copy the data requested by `offset` and `nbytes` into this buffer.
- `int offset` – The offset in bytes from the beginning of the file for the desired data.
- `int nbytes` – The number of bytes being requested.

The read function should return the number of bytes actually put into `buf`. A negative number indicates an error.

### Parameters

- `PatcherReadFn* ptr` – The function to be called to fetch a portion of the original data. This function will be called repeatedly and must be able to access the data randomly.
- `void* handle` – A handle to any information your read function will need to access the correct data.

### Return Values

None.

---

## PatchExists

```
bool PatchExists(const char* name, int& reqSize)
```

---

### Description

`PatchExists` determines whether the named module is modified by the patch in the current patch buffer. If so, the new size that the module will have after patching is returned in `reqSize`. The game can use that size to allocate a buffer to hold the result when applying the patch with `PatchItem`.

### Parameters

- `const char* name` – The name of the resource being checked.
- `int& reqSize` – If the patch exists, `reqSize` holds the size of the new version of the resource after patching with `PatchItem`.

### Return Values

- `true` – The named file or resource will be modified by the patch in the patch buffer.
- `false` – The named file or resource is not modified by the patch in the patch buffer.

---

## PatchItem

```
bool PatchItem(const char* name, char* toBuf, int toSize)  
bool PatchItem(char* toBuf, int toSize)
```

---

### Description

`PatchItem` applies a patch to a file or resource from the current patch buffer and stores it in the given buffer, `toBuf`. If the name of the item is given with the `name` parameter, `PatchItem` searches for the named item in the patch buffer. If the name is not given, the last item found with `PatchExists` is used, thus avoiding an extra search.

### Parameters

- `const char* name` – The name of an item to be patched. The current patch buffer is searched for the item. If it is not found, `PatchItem` returns `false`.
- `char* toBuf` – The output buffer for patching. The item will be patched directly into the buffer, which must be large enough to hold the entire new version of the item. The required buffer size can be obtained from an earlier call to `PatchExists`.
- `int toSize` – The size of the buffer pointed to by `toBuf`.

### Return Values

- `true` – The item was successfully patched and the result is stored in `toBuf`.
- `false` – The item could not be patched. There could be several causes for failure, listed below:
  - The item was not found in the patch. Either the item given by the `name` parameter is not in

the patch, or the call to `PatchItem` was not preceded by a call to `PatchExists`.

- The buffer was not large enough. Make sure the buffer is as large as indicated by `PatchExists`.
- The original source data did not match the data required to perform the patch (checksum mismatch). This error may be common during development, but should not happen in a released game. Make sure you have the same original file that was used to generate the patch.
- The original source data could not be read. Make sure that you are using a valid read function for accessing the original unmodified data, using either `SetSourceBuffer`, `SetSourceFile`, or `SetSourceReadFunction`.

## Methods Only on Playstation®2

---

### LoadModule

```
bool LoadModule(const char* name, char* buffer, int bufferSize)
bool LoadModule(char* buffer, int bufferSize)
```

---

#### Description

*The `LoadModule` method is only available on the Playstation®2.*

`LoadModule` loads the named or previously found SN Systems DLL module into the memory buffer, and relocates the code to enable functions in the DLL to be executed.

When a name is given with the `name` parameter, `LoadModule` will use `PatchItem` to patch the named module into the buffer. If there is no patch for the named module, the original is loaded into the buffer using the current read method. Hence, when a name parameter is given, `LoadModule` will work regardless of whether the module has been patched, assuming the read method can read the original module.

When no name parameter is given, the `LoadModule` method uses `PatchItem` to patch the last found item into the buffer.

Because `PatchItem` is used for the patching, please familiarize yourself with any notes or restrictions regarding its use.

#### Parameters

- `const char* name` – The name of the module to be patched and loaded. The module must be a valid SN Systems DLL file.
- `char* buffer` – The buffer into which the module will be patched and loaded.
- `int bufferSize` – The size of the buffer in bytes.

#### Return Values

- `true` – The module was successfully loaded into the buffer and the code relocated. Functions in that DLL can now be called.

- false – The load of the module failed. Either `PatchItem` failed (see `PatchItem` for more information), the specified module could not be read, or the specified module was not a valid DLL.

## FilePatcher Class

---

*The **FilePatcher** class is available only in the Windows PC version of the patch system, and is not needed by normal game applications.*

The **FilePatcher** class is a higher level subclass of the **PatchExtractor** class which allows the sequence of all files in the given patch buffer to be patched on disk. It is used by the patching application (see `LaunchPatcherApp` below) to patch the game directory.

When a **FilePatcher** is created, your application provides a directory to be patched, together with a buffer containing the patch data. All files in that directory which contain a patch can then be patched in turn using the `PatchNextFile` method.

Note that all the methods of **PatchExtractor** are also available.

## Methods

---

### Constructors

---

```
FilePatcher()
FilePatcher(const std::string& dstDir, char* patchData, int dataSize)
```

---

#### Description

The default constructor with no arguments is provided as a convenience to allow a **FilePatcher** to be instantiated anywhere, even before the patch data information is available. When the patch data information is available, the `Init` method must be used to enable file patching.

The constructor with arguments simply passes the arguments through to the `Init` function. See the documentation of the `Init` function for details. If this version of the **FilePatcher** constructor is used, you should check the `IsGood` method to make sure no errors occurred. See the documentation of `IsGood` under the **PatchExtractor** class for information.

As an illustration, note that the code

```
FilePatcher p;
if (p.Init(myDir, data, size)) ...
```

is equivalent to

```
FilePatcher p(myDir, data, size);
if (p.IsGood()) ...
```

---

### Init

---

```
bool Init(const std::string& dstDir, char* patchData, int dataSize)
```

---

#### Description

`Init` initializes this **FilePatcher**. The directory to be patched is specified with `dstDir`, and the patch data buffer and data size are both specified just as with a **PatchExtractor**.

### Parameters

- `const std::string& dstDir` – The directory to be patched. All files which need patching are patched in place in the given directory.
- `char* patchData` – The patch data buffer containing a prepared patch which was downloaded and prepared with the **Patcher** class.
- `int dataSize` – The size of the data in the `patchData` buffer. As with the **PatchExtractor** class, this size must be accurate in order for the checksums to be calculated properly.

### Return Values

- `true` – The patch data buffer contains good data and the **FilePatcher** is ready to patch the files.
- `false` – The call to `PatchExtractor::Init` failed. See the documentation of the `Init` function under **PatchExtractor** for further information.

---

### FindNextFile

```
int FindNextFile(std::string& filename)
```

---

#### Description

The `FindNextFile` method locates the next file to be patched in the patch data buffer, sets the **FilePatcher** internal pointer to that file, and returns information about the file for use in status displays.

#### Parameters

- `std::string& filename` – The name of the next file to be patched if found.

#### Return Values

- `1` – The next file was found successfully, and information about the file has been returned.
- `0` – No more files remain to be patched.
- `-1` – An error occurred while trying to parse the patch buffer. Because no integrity checks are performed at this stage, such an error would generally indicate that a previous call has failed, such as `Init`.

---

### PatchCurrentFile

```
int PatchNextFile(void)
```

---

#### Description

After a file has been found with `FindNextFile`, use `PatchCurrentFile` to apply the patch to the file on disk.

`PatchCurrentFile` uses the underlying **PatchExtractor** superclass to apply the patches, so it uses the same rules and can fail in the same instances as the `PatchItem` function described under **PatchExtractor**. Note that, like `PatchItem`, `PatchCurrentFile` will make use of the current read

method as set up by the `SetSourceBuffer`, `SetSourceFile`, or `SetSourceReadFunction` methods.

### *Parameters*

None. All information is provided to the `Init` function.

### *Return Values*

- 1 – The file was patched successfully.
- -1 – An error occurred. The error could be any of the following:
  - The earlier call to `Init` failed.
  - The directory given to the `Init` function does not exist.
  - Out of memory (unlikely with newer versions of Windows)
  - The call to `PatchItem` failed. See the description of `PatchItem` under **PatchExtractor** for more information.
  - Unable to write the output file. Because the directory has already been found to exist, this would generally indicate that the file is already open, such as would be the case if you tried to patch the currently running executable.



## Global Functions

---

This section describes simple functions which do not belong to any class.

### Functions only on Microsoft Windows

---

#### LaunchPatcherApp

```
bool LaunchPatcherApp(const char* host, const char* hostPath,
                      const char* dirToPatch, const char* updateApp = NULL)
```

---

#### Description

**LaunchPatcherApp** launches the **WinUpdater** application in a separate process for the purpose of patching all files contained in the given patch. It is assumed that your application has already determined the need for a patch before calling this function, therefore the version is not checked. Hence, if a patch is not available, it will be indicated as an error to the user.

If the function succeeds in launching **WinUpdater**, it returns `true` and your application should exit immediately. If the launch fails, **LaunchPatcherApp** will return `false`, and you likely have an error in your installation.

#### Parameters

- `const char* host` – Name of host holding the version description file. For example, *mygame.patch.eidos.com*.
- `const char* path` – Full path and file name of the patch version description file. The path must be complete. For example, */assets/version.txt*.
- `const char* dirToPatch` – The directory in which to apply the patch. This is generally the directory where your application is installed, for example “*C:\Program Files\MyGame*.”
- `const char* updateApp` – The full path (including file name) at which the updater application (usually **WinUpdater.exe**) can be found (e.g., “*C:\Program Files\MyGame\utils\WinUpdater.exe*”). If this argument is omitted or is `NULL`, **LaunchPatcherApp** looks in the current working directory for **WinUpdater.exe**.

#### Return Values

- `true` – The **WinUpdater** process was launched successfully, and your application should quit to allow it to be updated.
- `false` – An error occurred spawning the **WinUpdater** process. The most likely cause for this failure would be an inability to find **WinUpdater**, for example if `updateApp` were `NULL` and **WinUpdater** were not in the current working directory.

## Error Codes

---

This section describes error codes which can be returned by the `Patcher::GetPatchError` function.

- `PATCH_ERROR_NONE` – No error has occurred. Either the recent return did not specify an error, or the return value itself should have been a complete description of any error which occurred.
- `PATCH_ERROR_SOCKET` – The updater thread was unable to create a socket to connect to the server. This error is serious, and should not happen in the final game. It probably indicates a problem during development, such as being out of memory.
- `PATCH_ERROR_BAD_RESOURCE` – This error would be an internal error. It should not be possible to trigger with **Patcher** 1.2.
- `PATCH_ERROR_BUF_TOO_SMALL` – The buffer passed to the **Patcher** for receiving a patch, encrypting, or decrypting the patch (depending on what was last called), was too small. Make sure your application is checking the buffer return sizes from `TestCompletion`.
- `PATCH_ERROR_PARSE_ERROR` – The Patcher was unable to successfully parse the answer from the server. This error could indicate a server compatibility problem, such as the lack of a `Content-Length` field in the HTTP response header. See the section *Serving the Patch* in the document *Generating and Serving Patches* for more information.
- `PATCH_ERROR_CRYPT_INIT` – This error indicates a failure to initialize the encryption and/or decryption system for the current platform. For **Patcher** version 1.2, this error can only occur on the Playstation®2, and indicates a bad status return from `sceDNAS2InstInit_mc`, which should never occur if the application has been built correctly.
- `PATCH_ERROR_ENCRYPT_SIZE` – This error indicates that the encrypt system for this platform was unable to calculate the final size of the encrypted data. For **Patcher** 1.2, this error can only occur on the Playstation®2, and indicates that DNAS-inst was unable to calculate the size of the personalized data for storage on the memory card. Such failure may indicate that the raw data downloaded is corrupt or was not properly authored with the DNAS authoring system.
- `PATCH_ERROR_ENCRYPT` – This error indicates that the encrypt system for this platform was unable to encrypt the data. For **Patcher** 1.2, this error can only occur on the Playstation®2, and indicates that DNAS-inst was unable to personalize the data for storage on the memory card. Such failure may indicate that the raw data downloaded is corrupt or was not properly authored with the DNAS authoring system.
- `PATCH_ERROR_DECRYPT_SIZE` – This error indicates that the decrypt system for this platform was unable to calculate the final size of the decrypted data in a call to `PrepareData`. For **Patcher** 1.2, this error can only occur on the Playstation®2, and indicates that DNAS-inst was unable to calculate the size of the decrypted data in preparation for patching. Such failure may indicate that the encrypted (personalized) data is corrupt. If the data was read from the memory card, this could indicate a problem reading the data from the memory card. If the data was just downloaded successfully, this error should not happen unless there is memory corruption, since it would have just been personalized during the download process; so make sure you are checking the status of the download.
- `PATCH_ERROR_DECRYPT` – This error indicates that the decrypt system for this platform was unable to

decrypt the patch in a call to `PrepareData`. For **Patcher 1.2**, this error can only occur on the Playstation®2, and indicates that DNAS-inst was unable to decrypt the patch in preparation for patching. Such failure may indicate that the encrypted (personalized) data is corrupt. If the data was read from the memory card, this could indicate a problem reading the data from the memory card. If the data was just downloaded successfully, this error should not happen unless there is memory corruption, since it would have just been personalized during the download process; so make sure you are checking the status of the download.

- `PATCH_ERROR_REDIRECT` – This error indicates that the server returned an HTTP 3xx code indicating that the page has been moved and the client should re-request from a new URL. The Patcher does not support such redirection, so if this error occurs contact the system administrator for the server and make sure it does not redirect the patch requests.
- `PATCH_ERROR_HTTP_CLIENT` – This error indicates that the server returned an HTTP 4xx code other than 404 (file not found). Such an error would indicate a problem with the HTTP request from the client; once a game has been tested, this error should never occur. Should it occur, it would probably indicate memory corruption or some other serious client-side error which should be debugged.
- `PATCH_ERROR_HTTP_SERVER` – This error indicates that the server returned a code indicating an internal server error. If this error should ever occur, contact the administrator of the patch server.
- `PATCH_ERROR_UNRECOGNIZED_RESPONSE` – This error indicates that the server returned an HTTP 1xx or 2xx code which is not recognized by the patch server. If this error should ever occur, contact your patch server administrator and have him or her make sure that a successful response from the HTTP server would only ever be 200 (OK).
- `PATCH_ERROR_HOSTNOTFOUND` – This error indicates that the **Patcher** was unable to obtain an IP address for the host name of the patch server. It normally indicates a transient problem with the internet service provider, such as an error with their nameserver. If your application receives this error, it could try other server names from a list before failing. Because this error will occasionally happen with end users, it is important that your application handle it gracefully, letting the user know what went wrong and possibly allowing them to try again.
- `PATCH_ERROR_NOCONNECT` – This error indicates that a connection could not be made with the patch server. The server or an intermediate node on the internet may be down. Because this error will occasionally happen with end users, it is important that your application handle it gracefully, letting the user know what went wrong and possibly allowing them to try again.
- `PATCH_ERROR_SENDFAIL` – This error indicates that an attempt to send a request to the server failed. Such a failure could indicate that the server went down after connection or otherwise dropped the socket, or that some other network failure has occurred. Because this error will occasionally happen with end users, it is important that your application handle it gracefully, letting the user know what went wrong and possibly allowing them to try again.
- `PATCH_ERROR_TIMEOUT` – This error will not occur with **Patcher 1.2**, but indicates a timeout receiving data from the server.
- `PATCH_ERROR_POLLERROR` – This error indicates that an error occurred while polling the data receiving socket to see if data is available. This error should not occur. If it does, it may indicate an application programming error which caused the socket to become invalid (for example, closing the

network down while the **Patcher** is downloading).

- `PATCH_ERROR_RECVERROR` – This error indicates a problem reading from the receiving socket. It may indicate that the server dropped the connection for some reason, or became unreachable. Because such an error could occur in the completed game, it is important to either let the user know and allow them to try again, or to retry silently. The user should also have the option to stop trying, because the server may just not be available at the moment.
- `PATCH_ERROR_FILE_NOT_FOUND` – This error indicates that the selected patch data is not available on the server. Note that if the version definition file (VDF) is not found, this error is not returned, but rather the **Patcher** indicates a patch is not available. Hence, this error indicates that the patch described in the VDF is not on the server. Contact the administrator of your patch server to make sure that all patch information is in the correct place.