

Eidos Patch System

Client-Side Programming Guide for Playstation®2

Version 1.2

Updated April 6, 2005

Prepared by Pinniped Software

Pinniped Software



game and internet programming

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

Table of Contents

Overview.....	1
Restrictions.....	1
Memory Allocation Considerations.....	1
Integrating the Patch System.....	1
Required Modules.....	1
Working with DNAS.....	1
Memory Allocation.....	1
Satisfying Linker References.....	2
Loading the Patch System DLLs.....	2
Asynchronous Operations.....	3
Patch Discovery and Download.....	3
Patcher Creation and Destruction.....	3
Memory Card Load.....	4
Version Check and Patch Discovery.....	4
Determining Whether a Patch is Mandatory.....	5
Displaying a Description of the Patch.....	5
Downloading and Saving a Patch.....	5
Preparing the Patch for Use.....	6
Unloading the Patcher DLL.....	6
Applying Patches.....	6
PatchExtractor Basics.....	6
The Read Function.....	7
Built-In Read Functions.....	8
Loading Modified DLLs.....	8

Eidos Patch System

Client-Side Programming Guide for Playstation®2

Version 1.2

Overview

This document describes how to integrate client-side patching for the Eidos Patch System into your Playstation®2 title.

Restrictions

- This version of the patching system is written specifically to work with the SN Systems relocatable code (DLL—Dynamic Link Library) system. It is not currently compatible with other development environments, although a port may not be difficult.
- This version of the patching system only provides C++ interfaces.

Memory Allocation Considerations

Because every title uses its own memory allocation scheme, and because it may be desirable to use non-dynamically allocated memory in some instances (such as locating a DLL with the linker file instead of allocating memory for it), the patch system does no memory allocation on its own. Instead, it provides required memory sizes back to your application so that your application can manage the memory itself.

When designing the portion of your title that will use the patch system, you should consider the possibility of memory fragmentation, especially for loading and unloading of DLLs.

Integrating the Patch System

Required Modules

The patch download system uses the Sony Network Socket Library (functions beginning with *sceInsock*), and needs the IRX modules required for that library to be loaded already. In addition, the network must have been started with a good configuration. See the Sony-provided *Playstation®2 EE Library Overview for Network Libraries* document for more details.

Finally, the patch downloader uses the memory card version of the DNAS-inst library for decoding the patches.

Working with DNAS

When a patch is downloaded from the network, it is downloaded in a DNAS authored form, and DNAS-inst is used to *personalize* (encrypt using parameters specific to the currently running machine) the patch, and then to decrypt it into a usable format.

Memory Allocation

Even though the patch system does not allocate memory itself, it does make use of the Sony Network Socket Library (*sceInsock*) which performs its own memory allocation. Thus, if you want to make sure

all memory allocations go through your own system, you should use the functions `sceInsockSetMallocFunction` and `sceInsockSetFreeFunction` before starting the patch downloader. Make sure that your alloc/free functions are thread-safe, however, because the `sceInsock` functions will be called from a separate thread.

Satisfying Linker References

By default (which your application can change if you desire), the patch system is separated into two relocatable DLL (.rel) files, which of course must be loaded before your application can call the patch system functions.

The SN Systems DLL system provides the flexible ability to have a DLL or main program reference any symbol exported from another DLL or the main program. Minimizing the memory footprint of your title requires you to pay a little attention to which functions are defined in which modules, so as not to duplicate loaded code and so as to allow unneeded code to be unloaded at the appropriate time.

The tool you will need to use to examine which symbols are needed by which modules is `ps2dllcheck`, which is documented in the *Power User's Guide to ProDG for Playstation®2 Build Tools* document provided with your SN Systems ProDG distribution. Use this tool not only in your normal build procedure to check for missing symbols, but also to examine each DLL to see which external modules it requires. If a DLL requires modules which are not used anywhere outside the DLL, it makes sense to link those modules into that DLL, so that they may be unloaded when the DLL is unloaded.

Conversely, if a module is required by a DLL but is also used outside the DLL (for example, a common system function such as `strncpy`), you should link it into the main program or into another DLL which stays resident as long as the function is needed.

A safe default for the Patch system is to link the DNAS-inst library (`dnas_mc.a`) with the patch downloader DLL (the PatchLib project, which creates `PatchLib.rel`), and to export all the other symbols it needs from the main program or other DLLs. The patch application DLL (the PatchExt project, which creates `PatchExt.rel`) should also get most or all of its required symbols from the main program, except possibly the zlib library (`zlib.lib`), which should be linked into `PatchExt.rel` if you are not using it elsewhere in your program.

Loading the Patch System DLLs

It is recommended that you load the patch extractor, `PatchExt.rel`, first, followed by the patch downloader, `PatchLib.rel`. While neither DLL depends directly on the other, your application will usually need to make use of the extractor while the downloader is still loaded, and also after it has been unloaded. So assuming your application will be able to reuse the patch downloader memory after unloading, it is best to do the operations in the following order:

1. Start network and perform DNAS-net authentication
2. Load `PatchExt.rel` (contains **PatchExtractor**)
3. Load `PatchLib.rel` (contains **Patcher**)
4. Use **Patcher** to download patches from memory card and/or the network, store them, and decrypt (prepare) them.
5. Unload `PatchLib.rel`

6. Use **PatchExtractor** to load and patch all patchable DLL modules.
7. Use **PatchExtractor** to patch data.

In some games, the patch extractor will remain resident during the entire game to allow data to be patched as the player progresses. In those games, you can optionally compile and link the patch extractor modules directly into your application without creating another DLL. Alternatively, some games which load all data at the beginning of a level may choose to unload the patch extractor and free the memory containing patch data.

Asynchronous Operations

The patch system is designed to work as part of your main game loop, without the need for your application to explicitly create download threads. As such, the more time-consuming operations, such as network reads, are performed asynchronously using internally managed threads. In order for the patch system to be able to manage this thread, a few requirements are levied on your application:

- Each time through the main loop, you need to call the `Patcher::Update` method.
- If you have an outstanding asynchronous call, you can test for completion of the function using the `Patcher::TestCompletion` method.
- Make sure you assign the priority of the thread wisely. Because the PS2 does not use preemptive multitasking, it is important that you get the priorities correct. The `Patcher::Update` function will surrender time from the thread in which it is called (normally your title's main thread). It is important that you do not have other threads set up at a higher priority which may prevent the patcher network thread from running. Because the thread spends most of its time in a wait state (reading), it is generally safe to assign it a high priority without it adversely affecting your game.

Patch Discovery and Download

Both patch discovery and patch download are accomplished with the **Patcher** class, contained in `PatchLib.rel`. This section serves as an overview of the use of the **Patcher** class. For detailed information on the methods, see the *Eidos Patch System for Playstation 2 Client-Side Reference Manual*.

Patcher Creation and Destruction

A **Patcher** object is never created directly by the application. Rather, a single instance of a **Patcher** object is created by the static function `Patcher::Create()`, which takes as its only parameter a priority to be used for the background network downloading thread:

```
Patcher::Status stat = Patcher::Create(prio);
```

Likewise, when you are finished using the patch downloader, it can be destroyed with the `Destroy` function:

```
Patcher::Destroy();
```

Before the singleton **Patcher** is created, or after it is destroyed, an error will occur if you attempt to make any **Patcher** calls.

Memory Card Load

If your title is saving patches to the memory card to save download time, you should check the memory card for a saved patch before checking the network. The patch system provides no memory card access functions, so it is up to your application to load the patch from the memory card into RAM. Once the patch is in RAM, you will need to decrypt it.

First, tell the patch system about the preloaded patch:

```
Patcher* p = Patcher::GetPatcher();  
p->SetPatch(buffer, bufferSize);
```

Next, you need to decrypt the patch. To start the process, use:

```
p->PreparePatch();
```

`PreparePatch` is an asynchronous routine, so you will need to wait for it to complete by calling `TestCompletion` each time through your main loop until it is done.

Once `TestCompletion` indicates the decryption is complete, you will need to get the version number from the patch in order to perform the network version test. This is done using the patch extractor. Note that creating a **PatchExtractor** requires the clear (unencrypted) size of your data:

```
Patcher::CompletionInfo info;  
if (p->TestCompletion(info) == Patcher::PATCH_STATUS_OK)  
{  
    PatchExtractor extractor(buffer, info.clearSize);  
    if (extractor.IsGood())  
        version = extractor.QueryVersion();  
}
```

If `extractor.IsGood()` returns false, there was a problem with the patch, in which case the patch file should be deleted and you should act as if it weren't there.

Version Check and Patch Discovery

Your game will have a version number at this point, either from the original shipped game, or from the patch loaded from the memory card. That version number is used to query the network as to whether your game needs to be patched:

```
p->BeginVersionCheck(VDF_HOST_NAME, VDF_PATH, version);
```

When `TestCompletion` indicates the check is done, a size of zero for the required buffer size (`info.reqBufSize`) indicates that there is no patch needed, and a positive size indicates you must download a patch. If a patch download is needed, `info.reqBufSize` is the buffer size you will need for downloading the patch.

```
Patcher::CompletionInfo info;  
if (p->TestCompletion(info) == Patcher::PATCH_STATUS_OK)  
{  
    if (info.reqBufSize)  
    {  
        // A patch is needed. Allocate or otherwise  
        // acquire a buffer of info.reqBufSize bytes,  
        // and let the user know you are about to
```



```

        // download a patch.
    }
}

```

Determining Whether a Patch is Mandatory

When the `CompletionInfo` structure indicates a patch is available with a positive `info.reqBufSize`, it will also contain a flag indicating whether this patch is mandatory or not:

```

if (info.reqBufSize)
{
    // Display a different user message depending on whether the
    // patch is mandatory
    if (info.mandatory)
    {
        // Let user know the patch is required to continue...
    }
}

```

If a value for whether the patch is mandatory is not specified in the version description file, it is indicated as mandatory by default.

The actual behavior of the game and interpretation of the mandatory flag is left up to the game, but in general a patch flagged as mandatory would trigger the game to indicate to the user that they must download and install the patch to continue.

A list of non-mandatory patches could also be presented to the user to allow the user to select which ones to install.

Displaying a Description of the Patch

If a patch is available, you can download a description of it from the server if one has been specified in the version description file. Because the description is downloaded from the host serving the patch, it is downloaded asynchronously:

```

p->DownloadDescription(descbuf, bufsize);

```

Use `TestCompletion` to wait until the description is downloaded. Because the description is a file, it can be in whatever format your application desires. If the entire description does not fit in your buffer, it is truncated at `bufsize-1` bytes.

Downloading and Saving a Patch

If a patch is available for your game, you begin downloading it once you have a buffer large enough to hold it in memory:

```

p->DownloadPatch(buffer, bufferSize);

```

As with checking the version, you use `TestCompletion` to wait until the patch is fully downloaded. Once the download is complete, the patch is already personalized (encrypted) with DNAS, and may be stored to the memory card. The `storeSize` member of **CompletionInfo** indicates how much data needs to be stored on the memory card.

```

Patcher::CompletionInfo info;
if (p->TestCompletion(info) == Patcher::PATCH_STATUS_OK)
{
    // ...
}

```

```

// We have a patch ready to be stored to the memory
// card. Hand it off to the application's memory
// card storage system.
OurAppStoreBufferToMemoryCard(buffer, info.storeSize);
}

```

Preparing the Patch for Use

After a patch has been stored on the memory card, you need to prepare it for use by decrypting it in place, as described above under *Memory Card Load*:

```
p->PreparePatch();
```

When `TestCompletion` indicates it is finished, you will have a decrypted patch in memory ready for use by the patch extractor. Remember to save the size of the decrypted patch (`info.clearSize`) to initialize the **PatchExtractor** object.

Unloading the Patcher DLL

Once you have either determined you do not need a patch, or you have a valid patch in memory, you can unload the patch downloading DLL and free or reuse the memory it occupied:

```

Patcher::Destroy();

// patchLibMemory is where the PatchLib.rel DLL was loaded
snDllUnload( patchLibMemory );

// the memory at patchLibMemory can now be reclaimed by
// the application

```

Make sure that you do not delete the buffer in which the patch itself is stored, because it will need to be used by the patch extractor.

Applying Patches

Patches are applied to individual resources, or files, using an instance of the **PatchExtractor** class associated with a fully decrypted patch buffer as prepared from a download or from the memory card. Multiple **PatchExtractor** instances may be used with different buffers; for example, you could have separate patches for resources needed only at startup (DLLs, preloaded assets, etc.) and for resources which will be loaded dynamically during gameplay.

PatchExtractor Basics

As described briefly above, you instantiate a **PatchExtractor** for a downloaded patch, and then use it to extract individual patched resources. When you instantiate or initialize the **PatchExtractor**, it not only saves the buffer and size you provide, but also calculates an MD5 signature for the patch buffer, which is compared to the signature stored in the patch. If the signature does not match, the buffer is flagged as being no good, and patches cannot be extracted. A sample of creating an extractor and checking its status was given under the *Memory Card Load* section above.

When extracting an item, it is referred to by the disk file name it had when generating the patch, including the sub-directory name (using forward-slash syntax). For example:

```
extractor.PatchItem("texture/foo.img", itemBuf, itemSize);
```

Normally, you will check whether a patch exists before patching, and get the size required to receive it:

```
int fooSize;
if (extractor.PatchExists("texture/foo.img", fooSize)
{
    char* fooBuf = AllocAligned(fooSize, 64);
    extractor.PatchItem(fooBuf, fooSize);
}
```

Note that here `PatchItem` is called without a name, in which case it uses the last found item (from `PatchExists`). You could also use the version that takes the item name, as in the previous example; however, in that case both `PatchExists` and `PatchItem` would search the table of contents for the item, which could slow things down.

The Read Function

A patch is a series of modifications made to an original resource, and as such, needs to access the original unmodified resource when being applied. Unfortunately, the method used for patching precludes patching in place, at least for the current version. Hence, when applying a patch, the patch extractor needs to “read” the original source data in order to modify it.

Because each game uses different schemes to load assets, the **PatchExtractor** uses a user-definable *read function* to define the read method for the original source data. Two internal read functions are provided for convenience (direct from disk file or from an in-memory buffer), in the case where a custom function is not needed.

Here is a sample read function definition for reading out of a memory buffer, with all error handling removed for simplicity:

```
int MyRead(void* handle, char* buf, int offset, int nbytes)
{
    const char* p = (const char*)handle;
    memcpy(buf, &p[offset], nbytes);
    return nbytes;
}
```

And here is how it might be used to read a patch:

```
char* originalBuf;
PatchExtractor extractor(patchBuf, patchSize);
[...] // Assume originalBuf and extractor are properly
[...] // allocated and initialized

// Fetch the asset into your buffer using your own scheme
int originalSize = MyFetchAsset("myasset", originalBuf);

// Set the source read function to your function
extractor.SetSourceReadFunction(MyRead, originalBuf);

// Extract the item into newBuf
extractor.PatchItem("myasset", newBuf, newBufSize);

// Unlock/close asset
MyReleaseAsset(originalBuf);
```

Note that the read function takes a `void* handle` parameter which is provided by your application to access your private data. Note also that the read function does not get the asset name, and is called repeatedly for a given asset, so if your read function requires the asset to be opened (for example, if it is a file, or if it needs a handle into your asset system), you will need to do that before patching the item. Likewise, you may need to close your item after patching. Finally, note that the read function must be able to access the data in the asset randomly using the given offset.

Built-In Read Functions

As mentioned above, there are two built-in read functions, which are set using **PatchExtractor** methods. If your asset is a disk file which should be read directly from the DVD drive, it can be accessed by the default read method. You can reset the read function back to this read method with `PatchExtractor::SetSourceFile()`. If you wish to use a memory buffer as given in the above example, use `PatchExtractor::SetSourceBuffer()` rather than writing your own custom read function. The different read methods may be changed freely with different assets as in the sample below.

```
extractor.PatchItem("item1", item1Buf, item1BufSize);

extractor.SetSourceBuffer(myBuffer);
extractor.PatchItem("item2", item2Buf, item2BufSize);

extractor.SetSourceReadFunction(myRead, item3Handle);
extractor.PatchItem("item3", item3Buf, item3BufSize);
extractor.PatchItem("item4", item4Buf, item4BufSize);

extractor.SetSourceFile();
extractor.PatchItem("item5", item5Buf, item5BufSize);
```

In this example, items 1 and 5 are both read directly from DVD, item 2 is read from `myBuffer`, and items 3 and 4 are both read using the custom function `myRead`.

Loading Modified DLLs

A special patch function is available in the **PatchExtractor** to aid in patching DLL files. It will load the given DLL using the current read function and patch it and patch it if necessary. Hence, it can be used to substitute for the entire read-relocate sequence, regardless of whether your DLL files are located on disk or in your asset project file system. Once a patch is in memory, you can load all of your DLLs as follows.

```
extractor.LoadModule("sound.rel", sndDLLbuf, sndDLLbufsize);
extractor.LoadModule("network/play.rel", npBuf, npBufSize);
```

This technique is handy in the case where you have a fixed block of memory set aside for the DLLs in your linker file. If you are dynamically allocating the DLL memory, you can use `PatchExists` as in the above samples, and either `PatchItem` followed by your own call to `snDllLoaded`, or use `LoadModule` without a name. Note that in this case there must be a patch for `LoadModule` to work.

Below are examples using PatchExists with LoadModule:

```
static const char* dllName = "network/play.rel";
char* dllMem = NULL;
int dllSize = 0;

if (extractor.PatchExists(dllName, dllSize))
{
    dllMem = AllocAligned(dllSize, 128);
    extractor.LoadModule(dllMem, dllSize);
}
else
{
    LoadDLLIntoMemory(dllName, &dllMem);
    snDllLoaded(dllMem, 0);
}
```

Note that per the SN Systems documentation, the memory into which you load your DLL should be aligned to a 128-byte boundary.