Patch Manager

Preliminary

Developer Press

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

The Patch Manager

Contents

This chapter describes the Patch Manager and explains data-driven patching, a new patching model that you should use if you are writing patching code that is meant to run in Copland or in any subsequent Mac OS release.

A **patch** is a piece of code that intercepts the transaction between a client and a service. By using a patch you can monitor the use of this service or you can modify or replace the service. The Patch Manager is a shared library containing routines that you can use to obtain information about existing patches and to enable and disable patches. You should read this chapter if you are writing an application that must monitor, modify, or replace a routine residing in another fragment and if you cannot find any means to do so other than by patching that routine.

The Patch Manager is a new component of the Macintosh OS. It is intended to replace the routines used to install patches documented in the Trap Manager chapter of *Inside Macintosh: Operating System Utilities*. The programmatic patching model, referred to throughout this chapter, is based on the use of these routines. The Patch Manager is based on a new data-driven patching model that offers many advantages over the programmatic patching model used in system 7. Creating patches using the new model allows you to

- use a single patching model for head, tail, and surround patching
- patch any imported routine, not just operating system and toolbox routines
- create patches that have local as well as global effect
- enable and disable existing patches
- control the order in which patches execute
- obtain information about any currently installed patches

Copland also supports the patching API documented in the Trap Manager chapter of *InsideMacintosh:Operating System Utilities*. Note however, that this API will not be supported in future versions of the Mac OS. Thus, if you are certain that you need to patch, you might want to modify your patching code using the model described in this chapter to ensure compatibility with future operating system releases.

Although this chapter describes patching in some depth, you should rarely, if ever, need to use patches in an application. Historically, Apple has used patches to fix problems and augment routines in ROM code. The packaging of system services as a set of updatable shared libraries has eliminated this need for patching. Application developers have also used patching to get

information about system activity, to schedule services, or to customize the behavior of the system. Copland includes widely expanded notification services, new scheduling services, and many additional routines that you can use to customize the behavior of the system. Inasmuch as patching a routine is less economical and less dependable than using these newer services, you should seriously consider using the new services offered by Copland instead of patching system routines. However, because it is impossible to anticipate every need, the patching mechanism described in this chapter has been provided for your use.

About Patching and the Patch Manager

Patching a routine allows you to assume control every time the routine is called from a particular kernel process. If you assume control in order to do some preparatory processing before the routine executes, the patch is called a **head patch.** If you want to do additional processing after the routine executes, the patch is called a **tail patch**. If you want to do both, the patch is called a **surround patch.** It is also possible, though it is strongly discouraged, to write a **replacement patch,** which is executed instead of the routine being called.

When several applications patch the same routine, the result is a daisy chain of patches, or **patch chain,** with each patch in the chain executing in turn before the patched routine is called. Using the data-driven patching model defined for the Patch Manager, you can create every type of patch, specify when you want your patch to execute (relative to other patches in the patch chain), and have these patches automatically installed at the appropriate time —when the system starts up or when you launch the application, depending on the patch scope. Then, you can use Patch Manager functions

- to obtain information about all currently installed patches
- to determine which routines are being patched
- to enable or to disable patches

The following sections summarize the differences between programmatic and data-driven patching models, explains the special problems posed by patches with global effect, and examines the data-driven patching model in greater detail.

Programmatic and Data-Driven Patching

The patching model used in Copland differs markedly from the model used in system 7. This section describes these differences and examines the patching model used in Copland in greater detail.

The **programmatic patching** model defined for system 7 allows you to patch a system routine by replacing its address in the trap dispatch table with the address of your patch routine. If you need to call the patched routine from your patch, you are required to save the original address and then, in most instances, to write assembly language code that sets up the stack properly and then jumps to the saved address. Programmatic patching is in many ways limiting and costly to the programmer: It is limited to patching system software, it is difficult to implement, and it provides no control over the order in which patches execute. In addition, using this method it is not possible to examine the patch chain, which makes it next to impossible to identify and resolve patching conflicts.

The **data-driven patching** model defined for Copland and future Mac OS releases allows you to patch any routine by creating a data structure, called a **patch description structure,** for each patch you want the operating system to install. The patch description structure specifies a reference for the patch, a reference to the patched routine, the name of the patch, and other information used to control the execution of the patch. You store these data structures in a special fragment that is associated with your application fragment. When you launch your application, the Code Fragment Manager, Process Manager, and Patch Manager work together to load the patch code, prepare it for execution, and execute it at an appropriate time. You may store the patch routines in the fragment containing the patch description structures, but you are not required to do so.

Note
Because access to system routines in System 7 and in the first release of the PowerPC operating system is enabled through the use of a trap table, dispatched traps are extremely difficult to patch. Beginning with Copland, accessing system routines is the same as accessing any exported routine and the problem of dispatched traps is completely eliminated. ◆

The main advantage of the data-driven patching model is that it gives you more control over the execution of your patches at the same time that it does more of the programming work: Once you write the patch routine and create

the patch descriptions, you have no additional programming to do. The Code Fragment Manager and the Patch Manager assume all responsibility for installing your patches in the patch chain in the order that you specify, for advising you about any conflicting patches, and for maintaining information about the patches in a chain. In addition, the Patch Manager provides routines that you can use to obtain information about all installed patches for all processes on a machine and to enable or disable any patch.

Patch Scope

Using the programmatic patching model under System 7, patches can have local or global scope. If you want to assume control only when a routine is called by your application, you must create a patch that has local effect. This kind of patch is called a **local patch.** If you want to assume control when the routine is called by any kernel process, you must create a patch that has global effect. This kind of patch is called a **global patch.**

Under System 7, the distinction between local and global patches lies in whether the patches are installed by INIT-type code or whether they are installed by an application. Patches installed by INIT-type code have global effect; patches installed by an application have local effect. A programmatic patch that has global scope is loaded in the system heap: the same instance of the patch and of any data initialized by the patch are visible and accessible to all processes that call the patched routine. The Copland runtime architecture cannot support INIT-type code and does not support global programmatic patches.

Under Copland, all patches are local patches that are installed within a particular kernel process. You can achieve global-effect patching by having the operating system install a local patch in all current processes. Any processes that are launched before you install the global-effect patch are not affected by the patch.

To install a patch that has global effect in Copland, you must create a shared library fragment that uses per-process instantiation and a special patch fragment that contains the patch description structure.The patch code can reside in either fragment. The operating system instantiates such a library for each kernel process, and the iteration of the local patch across all current processes creates a patch with global effect. Each process contains its own copy of the patch and of any data initialized by that patch.

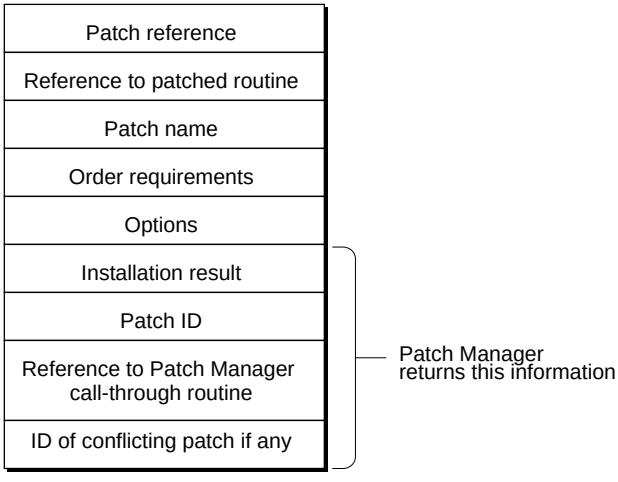
Because global-effect patches in the data-driven patching model are actually iterated local patches, if you create a patch that needs to share data or communicate with other instances of the patch, you must explicitly use standard Copland mechanisms for sharing and coordination. For more information, see NameOfBookToBeProvided, which describes these services.

Data-Driven Patching

This section describes the data-driven patching model in greater detail. It describes the data structure used to define a patch, explains the structure of the special fragment containing these structures, and discusses the ordering and execution of patches when multiple patches are applied to a single routine.

You use a data structure like the one shown in Figure 1-1 to describe each patch that you want to install.

Figure 1-1 The patch description structure



You initialize the patch description structure to contain the address of the patch and of the patched routine, the name of the patch, and any order requirements and options you specify for the patch.You use the order requirements field to describe the order in which you would like the patch to execute relative to

other patches. You use the options field to specify whether the patch is initially enabled or disabled and whether the patch must be successfully installed in order for the application to be launched.

In addition to its ID, every patch also has a name, that you assign to it when you create the patch. A patch name is composed of two parts: its signature and its type expressed as four-character literals. For example, `'WORD' 'SpCh'`.The signature must be the same as the registered creator code for the application or shared library installing the patch. The type part of the name is a four-character literal of your choice. If you are using programmatic patches, the Patch Manager will do the name assignment. If you are creating a data-driven patch and you do not assign it a name, the system will be unable to install the patch.

By using wildcard characters in lieu of either part of the patch name when specifying ordering requirements, you can cause your patch to be ordered relative to sets of patches. For example, you can ask that your patch execute before all patches installed by a given application, or you can ask that your patch execute after all patches of a specific type.

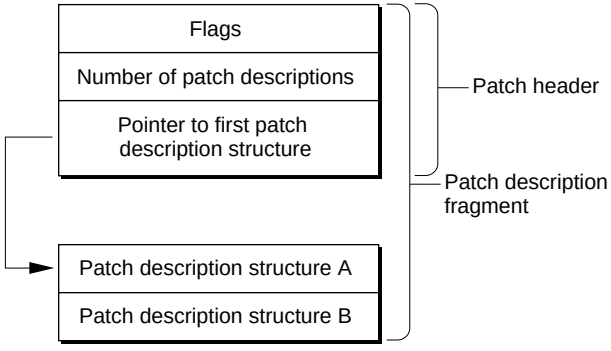
When your application or shared library is loaded, the Code Fragment Manager also loads and prepares the special fragment containing the patch descriptions. The Code Fragment Manager performs any required relocations and fills in actual addresses for the patch address and the patched routine address. The Patch Manager examines the ordering requirements you specified for each patch and attempts to place the patch in a patch chain according to those requirements. If it is able to do so, it returns a unique **patch ID** to identify the patch. If it is not able to do so, it returns an error code in the installation result field of the patch description structure and it also returns the ID of the patch that caused the installation of your patch to fail.

The Patch Description Fragment

In order to install a patch, you must create a special patch description fragment and store this fragment in the same file as that of the owning fragment. The patch description fragment should have the same name and usage code as that of the owning fragment, but it must have the update level 255. If any of these conditions are not met, the operating system will be unable to recognize and to install your patch.

The patch description fragment includes a patch header and one or more patch description structures. Figure 1-2 shows the structure of a patch description fragment that contains two patch description structures.

Figure 1-2 Patch description fragment.

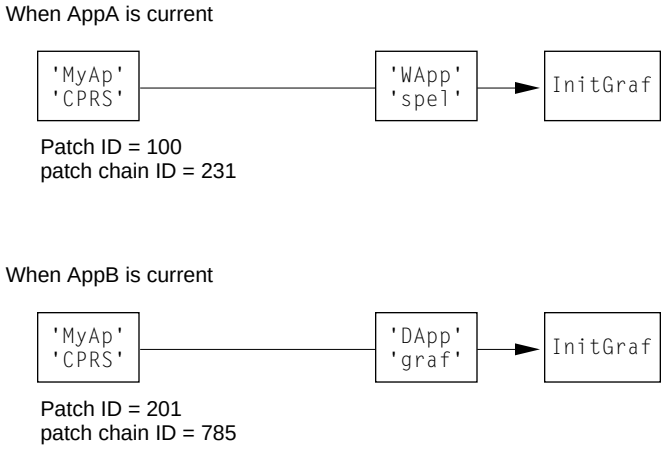


The patch header specifies the number of patch description structures in the fragment and points to the first one. The Code Fragment Manager and Patch Manager use the information provided by the patch header to locate all patch description structures in the fragment.The order in which patch description structures appear in the patch fragment affects the order in which the patches are installed but does not affect the order in which they execute. The order of execution is affected only by the values you specify for a patch's ordering requirements.

Applying Several Patches to the Same Routine

If several fragments patch the same routine, each patch is successively applied to the routine. Figure 1-3 shows two patches being applied to the routine `InitGraf`.

Figure 1-3 A patch chain



A **patch chain** is an ordered list of patches, all on the same instance of the same entry point. Figure 1-3 shows two patch chains for `InitGraf`. A patch chain includes a **root,** which is the routine being patched, and one or more patches. In Figure 1-3, the root of each patch chain is the routine `InitGraf`. The smallest patch chain contains one patch and the patch root.

Patch chains can contain local and global patches. The patch chains shown in Figure 1-3 include a global patch (`'MyAp' 'CPRS'`) and a local patch.

- The Code Fragment Manager instantiates a global patch for each kernel process that references the patched routine. This means that any data associated with the patch is instantiated in each kernel process. In addition, information about the patch, with the exception of the patch name, is also unique to the current kernel process. Note that the ID of the global patch and the ID of the patch chain to which it belongs is different, depending on the current kernel process.
- The local patches, `'WApp' 'spel'` and `'DApp' 'graf'` are also identified by unique patch IDs, even though they have different names.

Each patch chain is identified by a **patch chain ID.** This identifier is unique for each boot of the operating system. If two or more patch chains have the same root, the patch chain that executes is the patch chain that is current for a given kernel process. For example, when AppA is executing, patches `'MyAp' 'CPRS'`

and 'WApp' 'spel' are executed whenever the routine InitGraf is called. When AppB is executing, patches 'MyAp' 'CPRS' and 'DApp' 'graf' are executed whenever the routine InitGraf is called.

The order in which patches execute is determined by the ordering requirements you specify for each patch in the patch description structure. Otherwise, no ordering hierarchy exists. For example, global patches do not take precedence over local patches, and so on. Once the system has installed a patch in a patch chain, you can cause a patch not to execute by disabling it, but you cannot change its ordering requirements.

The Patch Manager provides routines that you can use to enable or disable a patch in a patch chain. It is important to understand that enabling a patch does not cause it to be added to the chain and that disabling a patch does not cause it to be removed from a chain. Enabling or disabling a patch simply determines whether the patch is executed when the patched routine is called.

The Structure of Patch Code

The structure of patch code used with data-driven patching is that of the surround patch. This type of patch

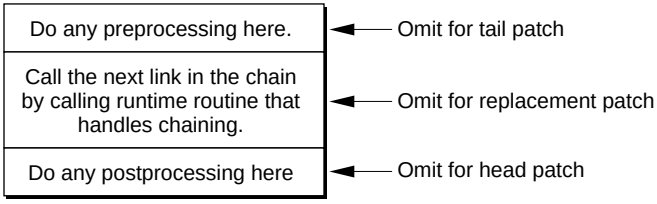
- does some preprocessing
- calls the patched routine
- does some post processing

Using the programmatic patching model, the call to the patched routine was the most difficult to implement. It involved saving the original address of the patched routine and then, in most cases, writing assembly-language code by means of which the stack was properly set up, and then a jump or branch was effected to the saved routine address.

The data-driven patching model replaces the troublesome second step with a simple standard runtime call to the Patch Manager that you write in a high-level language. This call tells the Patch Manager to call the next link in the patch chain and supplies the information it needs (return result type, routine parameters) to make the call. If the next link in the chain is a patch, the Patch Manager passes control to that patch. If the next link is the chain root, the patched routine actually executes.

The structure of data-driven patching code is shown in Figure 1-4.

Figure 1-4 The structure of a patch



As mentioned before, the patch code does not need to include any assembly code. It need only call the routine that handles the chaining. This routine ensures that the runtime environment is set up properly and that control passes smoothly to the next link in the chain. The Patch Manager returns the address of the routine that handles the chaining in the patch description structure.

A **replacement patch** is a patch that does not include a call to the chaining routine. To implement such a patch, you simply define a patch description structure for it, but you never call through the chaining routine. When you launch the application, the Patch Manager automatically installs the patch by placing it in the patch chain for the patched routine. It is important to understand that because a replacement patch does not call through the chaining routine, no patch following the replacement patch in the chain executes, including the routine being patched. For example, if `'MyAp' 'CPRS'` in Figure 1-3, is a replacement patch, neither `'WApp' 'spel'` nor `'DApp' 'graf'`, nor `InitGraf` are ever executed.

It is possible to specify order requirements that cause the replacement patch to be placed last in the chain (right before the patched routine). In this way all other patches in the patch chain execute before the replacement patch. However, it is not recommended that you do this, because setting up ordering requirements that are too strict is likely to result in your not being able to install your patch.

Order Requirements

You use the order requirements field of the patch description structure to specify when your patch should execute. The Patch Manager allows you to specify this order relative only to other patches, not to the patched routine. For example, you can specify that your patch execute

- before one patch and after another patch

- immediately before one patch and after another patch
- before one patch and immediately after another patch
- before a set of patches or after a set of patches

It is not a good idea to impose order requirements unless they are crucial to the performance of your code. If the Patch Manager cannot resolve conflicting order requirements, it cannot install your patch, and it returns this information to you in the patch description structure.

IMPORTANT

You cannot cause your patch to be installed by disabling the patch whose order requirements conflicts with yours. The Patch Manager must honor the ordering requirements of a patch even when that patch is disabled. Thus, the only way to eliminate ordering conflicts is to change your own ordering requirements. ▲

Limitations on Patching

If you use the data-driven patching model, you must observe the following restrictions:

- Patch code must be native.
- A library that owns a patch description fragment must use per-process instantiation.
- You can use only the generic routine that calls the next link in a patch chain to call patch code. That is, you can only call patch code as part of a patch chain.
- You cannot use the same patch code to patch two different routines.
- A fragment can have only one patch fragment associated with it.
- You must place the patch fragment and owning fragment in the same file.

Compatibility

The Patch Manager ensures that the trap patching API defined with System software 7 is supported in Copland. Support is limited to local patching. Table 1-1 lists the names of the functions that continue to be supported in Copland.

Table 1-1 Programmatic patching calls supported under Copland

Function
`GetTrapAddress`
`SetTrapAddress`
`NGetTrapAddress`
`NSetTrapAddress`
`GetOSTrapAddress`
`SetOSTrapAddress`
`GetToolTrapAddress`
`SetToolTrapAddress`
`GetToolboxTrapAddress`
`SetToolboxTrapAddress`
`GetTrapVector`

These functions are fully described in the Trap Manager chapter of *Inside Macintosh: Operating System Utilities*.

If you use these functions to install a programmatic patch, you can still use all the functions described in this chapter to obtain information patches (including the programmatic patch) and to enable or disable a patch.

Using the Patch Manager

The Patch Manager provides data structures that you use to describe your patch and the order in which you would like it to execute. The Patch Manager also provides functions that you can use to

- obtain a list of all the patches in a patch chain
- obtain information about any patch in a chain
- obtain a list of all the patch chains associated with a process
- determine the process to which a patch chain belongs
- determine the root of a patch chain
- determine the patch chain to which a routine belongs
- enable or disable any patch in a chain

This section explains how you use Patch Manager data structures and functions to create a patch, to control its execution, and to obtain information about currently installed patches.

Creating a Patch

This section explains the steps required to create a patch. Creating a patch involves

- Creating the source for the patch description fragment

  The source includes initializing a patch description header and one patch description for each patch.

  The patch description header specifies the number of patch descriptions you are going to include in the patch description file and provides a pointer to the array referencing the patch description structures. The patch description header must be the main symbol of the patch description fragment. The section "The Patch Header" on page 1-16 explains how you create a patch description header.

  The patch description specifies the patch, the patched routine, the patch name, and other options that govern the installation and execution of the

patch. The section "The Patch Description Structure" on page 1-17 explains how you initialize a patch description.

- Compiling and linking the source files containing the patch description header and the patch descriptions into a patch description fragment.

  This is a file having the same name and usage code as the fragment owning the patch description fragment, but an update level of 255.

- Write the patch routine and place it either in the patch description fragment or in any fragment that you are going to link with the patch description fragment.

The following two sections explain how you write a patch header and a patch description.

The Patch Header

Listing 1-1 shows the `PatchHeader` data type. The Patch Manager uses the first two fields for version control. You use the `count` field to specify the maximum number of patch description structures in the fragment. This number can exceed the actual number of patch descriptions; the Patch Manager strips trailing null pointers. You use the `patches` field to reference the patch descriptions.

Listing 1-1 The PatchHeader data type

```
struct PatchHeader {
    OSType                  tag;
    UInt32                  version;
    PatchHeaderOptions      flags;
    ItemCount               count;
    PatchDescriptionPtr *   patches;
};
typedef struct PatchHeader PatchHeader;
```

Listing 1-2 shows a sample patch header. It specifies the size of the array referencing the patch description structures and supplies a pointer to the first patch description structure.

Listing 1-2 Sample patch header

```
PatchDescription * gMyPatch[kMaxDesCount] ={gMyPatchX} /* Listing 1-4 */

PatchHeader gPatchData = {
    kPatchHeaderTag,
    kPatchHeaderVersion,
    kNilOptions,
    kMaxDescCount,
    gPatchDescription
    };
```

The system uses the `flags` field of the patch header to let you know whether it has failed to install any of your required or optional patches. If you are installing many patches, knowing that all your patches have been successfully installed can save you the trouble of examining each patch description structure to obtain the same information.

The Patch Description Structure

Listing 1-3 shows the `PatchHeader` data type. You use the first five fields to describe your patch: you provide a reference to the patched routine, a reference to the patch, a name for the patch, a constant value indicating the order in which you want the patch to execute relative to other patches, and a constant value indicating whether the patch should be initially enabled or disabled. The section "Order Requirements," beginning on page 1-12 explains how you specify a value for the field `thisPatchOrdering`.

The Patch Manager uses the last four fields to return information to you about the installation of the patch. It tells you whether the installation succeeded, it returns the ID of the patch, it provides a reference to the Patch Manager routine that handles the chaining, and, if your patch could not be installed because its ordering requirements conflicted with those of another patch, it returns the ID of the conflicting patch.

Listing 1-3 The PatchDescription data type

```
struct PatchDescription {
    PatchableProcPtr        originalRoutine;
    PatchableProcPtr        patchRoutine;
```

```
    PatchName               thisPatchName;
    PatchOrderRequirements  thisPatchOrdering;
    PatchOptions            installOptions;
    OSStatus                installResult;
    PatchID                 thisPatchID;
    PatchableProcPtr        thisCallThroughProc;
    PatchID                 rejectingPatchID;
};
typedef struct PatchDescription PatchDescription;
```

Listing 1-4 shows a sample patch description. It specifies the name of the patch to be `'wild',demo'`; it specifies that the patch can be placed anywhere in the patch chain and the it should be initially enabled.

Listing 1-4 Sample patch description

```
PatchDescription gMyPatchX = {
    &SysBeep,
    &MySysBeepPatch,
    {   'wild',
        'demo'},
    {   kNilOptions,
    {   kDoNotMatchAnyOrderedItemService,
        kDoNotMatchAnyOrderedItemSignature},
    {   kDoNotMatchAnyOrderedItemService,
        kDoNotMatchAnyOrderedItemSignature}},
    kPatchEnabledBit,
    paramErr,
    kInvalidID,
    NULL,
    kInvalidID
};
```

The initial values of the four output fields are not critical; however, using the ones shown in Listing 1-4 can help ensure that errors are detected.

Listing 1-5 shows a sample surround patch for the routine `SysBeep`. When the patch runs, it causes the menu bar to flash right before and right after the system beep sounds.

Listing 1-5 Sample patch code

```
void MySysBeepPatch(SInt16 duration)
{
// Preprocess
    FlashMenuBar(0);
    (void) DelayFor(kDurationMillisecond * 116);
    FlashMenuBar(0);

// This is where the call to the chaining routine goes

// Postprocess
    FlashMenuBar(0);
    (void) DelayFor(kDurationMillisecond * 116);
    FlashMenuBar(0);
}
```

Listing 1-6 shows a complete patch description fragment.

Listing 1-6 Patch Description Fragment

```
#include <Patches.h>
#include <CodeFragments.h>


#define AnyOrderedItemName {kMatchAnyOrderedItemService,
                            kMatchAnyOrderedItemSignature}
#define NoOrderedItemName  {kDoNotMatchAnyOrderedItemService,
                            kDoNotMatchAnyOrderedItemSignature}

#define DoNotCareItemOrder {kNilOptions, AnyOrderedItemName,
                            AnyOrderedItemName}


typedef OSStatus   (*GetIndexedSymbolPtr)  ( CFragConnectionID connID,
     SInt32symIndex,
     Str255symName,
     LogicalAddress *  symAddr,
     CFragSymbolClass *symClass );
```

```
OSStatus   MyGetIndexedSymbol  ( CFragConnectionID connID,
     SInt32symIndex,
     Str255symName,
     LogicalAddress *  symAddr,
     CFragSymbolClass *symClass );

enum {
   kMyPatchCount = 3
};

#define gMyGetIndexedSymbolPatchName   {'PTst', 'GSym'}

PatchDescription   gMyGetIndexedSymbolPatch= InitialPatchDescription
    (CFragGetIndexedSymbol,
     MyGetIndexedSymbol,
     gMyGetIndexedSymbolPatchName,
     DoNotCareItemOrder,
     (kPatchEnabledMask | kPatchOptionalMask) );

PatchDescription * gMyPatchDescriptions [kMyPatchCount] =
                                    { NULL, &gMyGetIndexedSymbolPatch };

PatchHeadergMyPatches  = { kPatchHeaderTag, kPatchHeaderVersion,
                    kNilOptions, kMyPatchCount, gMyPatchDescriptions };

OSStatus   MyGetIndexedSymbol  ( CFragConnectionID connID,
     SInt32symIndex,
     Str255symName,
     LogicalAddress *  symAddr,
     CFragSymbolClass *symClass )
{
   OSStatuserr;
   GetIndexedSymbolPtr NextGetIndexedSymbol=
                            gMyGetIndexedSymbolPatch.thisCallThroughProc;

   DebugStr ( (StringPtr) "\pIn MyGetIndexedSymbol" );

   err = (*NextGetIndexedSymbol) ( connID, symIndex, symName, symAddr,
                                                        symClass);
```

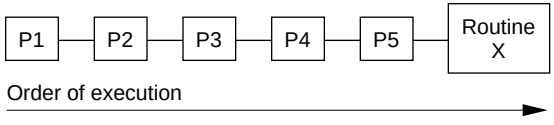
```
    return err;
}
```

[Discussion of Listing 1-6: To be done.]

Specifying Order Requirements

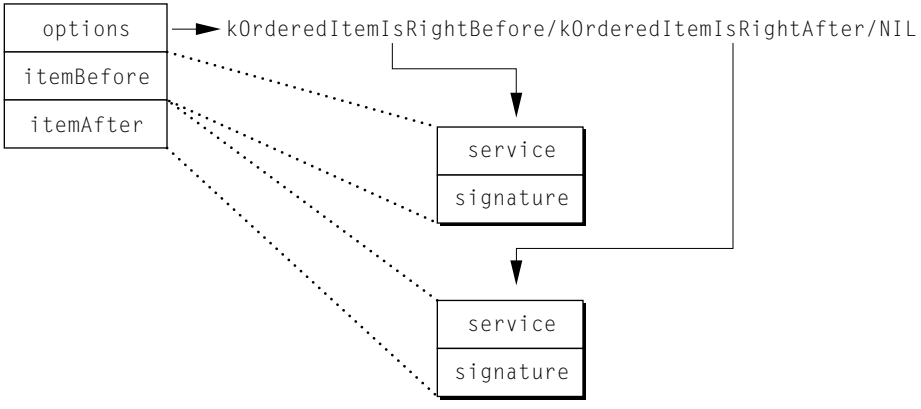
The field `thisPatchOrdering` in the patch description structure is a patch order requirements structure that you use to specify the order in which you want your patch to execute. It is easiest to explain how you use this structure by referring to a sample patch chain like the one shown in Figure 1-5.

Figure 1-5 Sample patch chain



The figure shows a set of five patches, P1 through P5, being applied to a routine. The structure you use to control the placement of your patch in that chain is shown in Figure 1-6.

Figure 1-6 Patch order requirements structure



The structure contains three fields. You use the field `itemBefore` to specify the name of the patch after which you want your patch to execute; for example, if your patch is P3 and you want it to execute after patch P1, you can specify the patch name of P1 in the `itemBefore` field when you create the patch. You use the field `itemAfter` to specify the name of the patch before which you want your patch to execute. For example, if your patch is P3 and you want it to execute before P4, you can specify the patch name of P4 in the `itemAfter` field when you create the patch.

You use the `options` field of the order requirements structure to make the selection specified with either of the other two fields more precise:

- If you want your patch to execute immediately after the patch given in the `itemBefore` field, specify the constant `kOrderedItemIsRightBefore` in the `options` field.
- If you want your patch to execute immediately before the patch given in the `itemAfter` field, specify the constant `kOrderedItemIsRightAfter` in the `options` field.

  Note that this scheme allows you to order your patch immediately after or immediately before another patch, but not both. It is best, however, unless the execution of your patch absolutely requires it, that you set the options field to `NIL`. This results in a looser ordering requirement and, consequently, in fewer ordering conflicts when the system installs your patch or other patches in the same patch chain.

You can also use the constant names listed in Table 1-2 in the `itemBefore` or `itemAfter` fields to specify your location relative to a *set* of patches.

Table 1-2 Wildcard specifiers

Constant name	Meaning
`kMatchAnyOrderedItemService`	Place my patch before/after every patch name whose signature field matches my signature field.
`kMatchAnyOrderedItemSignature`	Place my patch before/after every patch name whose service field matches that specified in my service field.
`kDoNotMatchAnyOrderedItemService`	I don't care where you place my patch.
`kDoNotMatchAnyOrderedItemSignature`	I don't care where you place my patch.

The following examples illustrate the use of the constant names shown in Table 1-2.

- To have your patch execute right before the patched routine, specify `kMatchAnyOrderedItemService` and `kMatchAnyOrderedItemSignature` in the `itemBefore` field. Specify `kDoNotMatchAnyORderedItemService` and `kDoNotMatchAnyOrderedItemSignature` for the `itemAfter` field.
- To have your patch execute first, specify `kMatchAnyOrderedItemService` and `kMatchAnyOrderedItemSignature` in the `itemAfter` field. Specify `kDoNotMatchAnyORderedItemService` and `kDoNotMatchAnyOrderedItemSignature` for the `itemBefore` field.
- To have your patch execute before a patch installed by application OtherApp, specify the application's creator code in the signature field of the `itemAfter` field, and set the service field to `kDoNotMatchAnyORderedItemService`.

▲ **WARNING**
The more restrictive you make the order requirements for your patch, the more likely they are to conflict with the order requirements of other patches. As a result, it will not be possible to install your patch or other conflicting patches. ▲

Creating a Local Patch

The steps required to create a local patch are listed in "Creating a Patch," beginning on page 1-15. Once you have created the patch description fragment, you must assign it the same name and creator as that of the owning fragment but an update level of 255. The owning fragment can be an application, a shared library, or any other kind of fragment except an update library.

Creating a Global Patch

The steps required to create a global patch are exactly the same as those for creating a local patch except that the owning fragment must be a shared library using per-process instantiation.

[To be done: Explain how the patch gets recognized and loaded.]

The patch defined by the patch description fragment is installed for all processes that are launched after the shared library is loaded.

If the patch code resides in the patch description fragment, the shared library fragment can be empty.

Creating a Patchable Shared Library

If you want to create a patchable shared library, it is important to make sure that your development system generates all internal calls to your exported routines as indirect calls. Some compilers and linkers do not call through transition vectors for routines in the same compilation unit. The data-driven patching mechanism depends upon the fact that patchable routines are accessed through transition vectors. If your compiler and linker do not call through a transition vector for an internally referenced routine and such a routine is patched, not all calls to the routine will see the patch.

Obtaining Information About Patches

The Patch Manager supports a variety of functions that return information about installed patches. This section offers a brief discussion of how these functions relate to one another and provides a more detailed discussion of the function used to return information about a patch.

Table 1-3 shows the input and output parameters of the Patch Manager functions that return information.

Table 1-3 Patch Manager functions that return information

Function	Input	Output
`GetPatchChainsInKernelProcess`	Process ID	Patch chain IDs
`GetPatchChainInformation`	Patch chain ID	Process ID Chain root
`GetPatchChainFromProcPtr`	Routine	Patch chain ID
`GetPatchFromProcPtr`	Routine	Patch ID
`GetPatchesInPatchChain`	Patch chain ID	List of patch IDs
`GetPatchInformation`	Patch ID	Patch chain ID Patch address Options Patch name

As you can see, given any single piece of information, you can use these routines to determine how that piece fits into the current patching scheme for all installed patches. For example, for any given process, you can determine the IDs of the patch chains associated with it and consequently the IDs of all the patches in the patch chains. For any given routine, you can determine whether it is included in a patch chain and, if so, the process to which the patch chain belongs.

You use the `GetPatchInformation` function to obtain information about a single patch; the function returns this information in a patch information structure. This includes the ID of the patch chain to which the patch belongs, the address of the patch, the name of the patch, its order requirements, and the options that are currently set for the patch. Current option settings are returned in the `patchOptions` field; these determine whether

- the patch is optional or required

  An optional patch is a patch that does not have to be installed in order for the application that references the patched routine to be launched. A required patch is a patch that has to be installed in order for the application to be launched.

- the patch is a data-driven patch or a programmatic patch

  In Copland, some applications might call routines that are patched programmatically. Such patches are included in the patch chain and you can use Patch Manager functions to obtain information about these patches, to enable them, and to disable them.

- the patch is currently enabled or disabled

  You can examine the `installOptions` field of the patch description structure to determine whether the patch is initially enabled or disabled.

Patch Manager Reference

Constants and Data Types

Option Bits Mask Enumeration

The option bits mask enumeration specifies possible values for the `installOptions` field of the patch description structure (page 1-32) and for the `patchOptions` field of the patch information structure (page 1-34). When you install a patch, you use the patch description structure to describe the patch. After the patch is installed, the Patch Manager returns information about its current state in a patch information structure.

You can use the bit OR operator to combine two or more of the following values.

```
enum {
    kPatchEnabledMask       =   (1L << kPatchEnabledBit),
    kPatchCompatibilityMask =   (1L << kPatchCompatibilityBit),
    kPatchOptionalMask      =   (1L << kPatchOptionalBit)
};
```

Enumerator descriptions

`kPatchEnabledMask` Before installing a patch, you use this bit in the `installOptions` field of the patch description structure to specify whether the patch is initially enabled. After installing the patch, you can use the `EnablePatch` function (page 1-36) and `DisablePatch` function (page 1-37) to enable and disable the patch.

When you examine the `patchOptions` field of the patch information structure, the setting of this bit indicates the initial state of the patch: The bit is set if the patch was enabled; the bit is clear if the patch was disabled. To obtain the current state of the patch, you must call the `GetPatchInformation` function (page 1-42).

`kPatchCompatibilityMask`

Do not use this bit in the `installOptions` field of the patch description structure.

If this bit is set in the `patchOptions` field of the patch information structure, it means that this is a programmatic patch. If this bit is clear, it means that this is a data-driven patch.

`kPatchOptionalMask` The setting of this bit in the `patchOptions` field of the patch description structure, tells the Code Fragment Manager how to proceed if it cannot load a patch. If this bit is set, it means the Code Fragment Manager should launch your application anyway. If this bit is clear, the Code Fragment Manager should not launch your application. In this case, it is your application's responsibility to notify the user of the cause of failure.

Wildcard Enumeration

You use the wildcard enumeration to specify values for the `service` and `signature` fields of the ordered item name structure (page 1-33). You use the ordered item name structure to specify values for the `itemBefore` and `itemAfter` fields of the order requirements structure (page 1-34). The effect of using wildcard order specifiers is explained in "Specifying Order Requirements" on page 1-21.

```
enum
{
    kMatchAnyOrderedItemService         =   (OrderedItemService)'****',
    kMatchAnyOrderedItemSignature       =   (OrderedItemSignature)'****',
    kDoNotMatchAnyOrderedItemService    =   (OrderedItemService)'----',
    kDoNotMatchAnyOrderedItemSignature  =   (OrderedItemSignature)'----'
};
```

`kMatchAnyOrderedItemService`

In the `service` field of the `itemBefore` field, this value causes your patch to execute after all other patches whose signature matches that specified in the `signature` field of the `itemBefore` field.

In the `service` field of the `itemAfter` field, this value causes your patch to execute before all other patches whose

signature matches that specified in the `signature` field of the `itemBefore` field.

`kMatchAnyOrderedItemSignature`

In the `signature` field of the `itemBefore` field, this value causes your patch to execute after all other patches whose service matches that specified in the `service` field of the `itemAfter` field.

In the `signature` field of the `itemAfter` file, this value causes your patch to execute before all other patches whose `service` field matches that specified in the `service` field of the `itemAfter` field.

`kDoNotMatchAnyOrderedItemService`

In the `service` field of the `itemBefore` field, this value means that you don't care if any patch executes before your patch.

In the `signature` field of the `itemAfter` field, this value means that you don't care if any patch executes after your patch.

`kDoNotMatchAnyOrderedItemSignature`

In the `service` field of the `itemBefore` field, this value means that you don't care if any patch executes before your patch.

In the `signature` field of the `itemAfter` field, this value means that you don't care if any patch executes after your patch.

Ordered Item Enumeration

You use the ordered item enumeration to specify a value for the `options` field of the order requirements structure (page 1-34). This value determines whether your patch is executed immediately before or immediately after another patch. Set this field to `NULL`, to indicate that the patch does not have execute right before or right after another patch.

```
enum
{
    kOrderedItemIsRightBefore   =   0x00000001,
    kOrderedItemIsRightAfter    =   0x00000002
};
```

`kOrderedItemIsRightBefore`
: The `itemBefore` field of the order requirements structure specifies the name of the patch that should execute before your patch. If you want that patch to execute immediately before your patch, specify this constant name for the `options` field of the order requirements structure.

`kOrderedItemIsRightAfter`
: The `itemAfter` field of the order requirements structure specifies the name of the patch that should execute after your patch. If you want that patch to execute immediately after your patch, specify this constant name for the `options` field of the order requirements structure.

Patch Header Flags Enumeration

The patch header flags enumeration is used by the Patch Manager to specify a value for the `flags` field of the patch header structure (page 1-31).

After the Code Fragment Manager has loaded your code fragment and patch fragment, and has installed your patches, it uses the `installResult` field of the patch description structure to let you know whether a particular patch was installed. At the same time, the Code Fragment Manager sets a bit in the `flags` field of the patch header structure to let you know whether *any* required or optional patch in a given patch chain set has failed. Thus, if you examine the `flags` field first and find either or both bits clear, you can save yourself the trouble of looking at the `installResult` field for each patch you have installed.

```
typedef OptionBits PatchHeaderOptions;
enum {
    kRequiredPatchErrorsMask    = 0x00000001,
    kOptionalPatchErrorsMask    = 0x00000002
};
```

Enumeration descriptions

`kRequiredPatchErrorsMask`
: If this bit is set, the system has failed to install one or more required patches.

`kOptionalPatchErrorsMask`
: If this bit is set, the system has failed to install one or more optional patches.

Patch Header Tag Enumeration

You use the patch header tag enumeration to specify values for the `tag` field and the `version` field of the patch header structure (page 1-31).

```
enum {
    kPatchHeaderTag     = 'Ptch'
    kPatchHeaderVersion = 1
};
```

Enumeration descriptions

`kPatchHeaderTag` Value for the `tag` field of the patch header structure.

`kPatchHeaderVersion` Value for the `version` field of the patch header structure.

Patch Header Structure

You use the patch header structure to specify the address and size of the array containing the patch descriptions for your local or global patches.

The `PatchHeader` data type defines a patch header structure.

```
struct PatchHeader {
    OSType                  tag;
    UInt32                  version;
    PatchHeaderOptions      flags;
    ItemCount               count;
    PatchDescriptionPtr *   patches;
};
typedef struct PatchHeader PatchHeader;
```

Field descriptions

`tag` The constant name `kPatchHeaderTag`.

`version` The constant name `kPatchHeaderVersion`.

`flags` This field is set by the Code Fragment Manager to one of the values defined by the patch header flags enumeration (page 1-30).

`count` The number of patch description structures in the array referenced by the `patches` field, described next.

`patches`	A pointer to an array of pointers to patch description structures that specify the local or global patches that you want to install.

Patch Description Structure

You use the patch description structure to specify a reference to your patch routine, a reference to the routine you want to patch, the name of your patch, and additional information about how you want the Patch Manager to execute your patch. The Patch Manager uses some fields in this structure to return information about the patch if it was successfully installed or to let you know about possible sources of conflict if it could not install the patch.

You must include one patch description structure for each patch you want to install. The `PatchDescription` data type defines a patch description structure.

```
struct PatchDescription {
    PatchableProcPtr        originalRoutine;
    PatchableProcPtr        patchRoutine;
    PatchName               thisPatchName;
    PatchOrderRequirements  thisPatchOrdering;
    PatchOptions            installOptions;
    OSStatus                installResult;
    PatchID                 thisPatchID;
    PatchableProcPtr        thisCallThroughProc;
    PatchID                 rejectingPatchID;
};
typedef struct PatchDescription PatchDescription;
```

Field descriptions

`originalRoutine`	A pointer to the routine you want to patch.
`patchRoutine`	A pointer to the patch routine.
`thisPatchName`	The name of your patch routine. You use the ordered item name structure (page 1-33) to specify the name of your patch.
`thisPatchOrdering`	The order in which you want your patch to execute. You use the order requirements structure (page 1-34) to specify whether your patch should execute before or after some other named patch.

You can specify `NULL` if you do not care when the pach executes.

`installOptions` A value given by the option bits mask enumeration (page 1-27) specifying whether the patch is initially enabled and whether the Code Fragment Manager should go ahead and load your application if it is unable to install your patch.

`installResult` The Patch Manager returns 0 in this field if the patch was successfully installed, or a nonzero result if it was not.

`thisPatchID` The Patch Manager returns the patch ID in this field if the patch was successfully installed.

`thisCallThroughProc`
The Patch Manager sets this field to the address of the routine you call to transfer control to the next routine in the patch chain.

`rejectingPatchID` The Patch Manager sets this field to the ID of a patch whose ordering requirements conflict with those you have specified for your patch. Disabling the conflicting patch does not solve ordering conflicts, but changing your ordering requirements might resolve the conflict.

If there are no conflicts, the Patch Manager sets this field to the constant value `kInvalidID`.

Ordered Item Name Structure

You use the ordered item name structure to specify values for the `itemBefore` and `itemAfter` fields of the order requirements structure (page 1-34). A patch must be uniquely named within a patch chain.

The `OrderedItemName` data type defines an ordered item name structure.

```
struct OrderedItemName {
    OrderedItemService      service;
    OrderedItemSignature    signature;
};
typedef struct OrderedItemName OrderedItemName, *OrderedItemNamePtr;
```

`service` A four-character literal that you define to identify a patch or one of the wildcard service enumerators (page 1-28).

`signature` A four-character literal that specifies the application's creator or one of the wildcard signature enumerators (page 1-28).

Order Requirements Structure

You use the order requirements structure to specify a value for the field `thisPatchOrdering` of the patch description structure (page 1-32). This field defines the relative order in which you want your patch to execute.

The use of this structure is explained in "Specifying Order Requirements" on page 1-21. The `OrderRequirements` data type defines the order requirements structure.

```
struct OrderRequirements {
    OrderedItemOptions  options;
    OrderedItemName     itemBefore;
    OrderedItemName     itemAfter;
};
typedef struct OrderRequirements OrderRequirements, *OrderRequirementsPtr;
```

`options` The value you specify for this field determines whether the item specified in the `itemBefore` field should come immediately before your patch or whether the item specified in the `itemAfter` field should come immediately after your patch. You specify one of these two values using the ordered item enumeration (page 1-29). If you do not need to specify either, use `kNilOptions`.

`itemBefore` The ordered item name structure (page 1-33) for the patch that should execute before your patch.

`itemAfter` The ordered item name structure (page 1-33) for the patch that should execute after your patch.

Patch Information Structure

The Patch Manager uses the patch information structure to return information to you about a patch specified by the `PatchID` parameter to the `GetPatchInformation` function (page 1-42).

The `PatchInformation` data type defines a patch information structure.

```
struct PatchInformation {
    PatchChainID            patchChain;
    PatchableProcPtr        patchingRoutine;
    PatchOptions            patchOptions;
    PatchName               patchName;
    PatchOrderRequirements  patchOrder;
};
typedef struct PatchInformation PatchInformation, *PatchInformationPtr;
```

Field descriptions

`patchChain`	The patch chain ID of the patch chain to which the patch belongs.
`patchingRoutine`	The address of the patch routine.
`PatchOptions`	A value given by the option bits enumeration (page 1-27) specifying whether the patch is currently enabled, whether it is a programmatic or data-driven patch, and whether the code fragment loader can launch an application even when it cannot install the patch.
`patchName`	The name of the patch. If this is a programmatic patch, the name is one assigned by the Patch Manager.
`patchOrder`	An order requirements structure (page 1-34) specifying the names of patches whose execution must precede or follow that of this patch.

Patch Chain Information Structure

You call the `GetPatchChainInformation` function (page 1-39) to determine the address of the routine that is being patched and the kernel process ID of the process that contains the patched routine. The `GetPatchChainInformation` function returns this information in a patch chain information structure.

The `PatchChainInformation` data type defines a patch chain information structure.

```
struct PatchChainInformation {
    KernelProcessID     kernelProcess;
    PatchableProcPtr    chainRoot;
};
typedef struct PatchChainInformation PatchChainInformation,
                                     *PatchChainInformationPtr;
```

Field descriptions

`kernelProcess`	The ID of the process containing the patched routine.
`chainRoot`	A reference to the routine being patched.

Patch Manager Functions

Enabling and Disabling Patches

When you define a patch using the patch description structure (page 1-32), you use the `installOptions` field to specify whether the patch should be enabled or disabled by default. You use the functions described in this section to change that default setting.

EnablePatch

Enables a patch.

```
OSStatus EnablePatch (PatchID thePatch);
```

`thePatch` The ID of the patch being enabled.

DISCUSSION

Enabling a patch causes the Patch Manager to execute the patch when the routine being patched is called. The order in which the patch executes depends upon ordering constraints specified by the field `thisPatchOrdering` of the patch description structure for this patch.

Any client that can obtain a patch ID can enable or disable a patch.

SEE ALSO

Use the `DisablePatch` function (page 1-37) to disable a patch.

Use the `GetPatchInformation` function (page 1-42) to determine whether a patch is enabled or disabled.

You use the patch description structure (page 1-32) to specify any ordering requirements for a patch.

DisablePatch

Disables a patch.

```
OSStatus DisablePatch (PatchID thePatch);
```

`thePatch` The ID of the patch being disabled.

DISCUSSION

Disabling a patch causes the Patch Manager not to execute the patch when the routine being patched is called.

Disabling a patch is not the same as removing a patch from the patch chain. For example, if any ordering conflicts exist, they remain even though the patch causing the conflict is disabled.

SEE ALSO

Use the `EnablePatch` function (page 1-36) to enable a patch.

Use the `GetPatchInformation` function (page 1-42) to determine whether a patch is enabled or disabled.

Obtaining Information About Patch Chains

You use the functions described in this section to obtain a list of patch chains associated with a process and to obtain information about a particular patch chain.

GetPatchChainsInKernelProcess

Returns a list of all patch chains associated with a process.

```
OSStatus GetPatchChainsInKernelProcess(KernelProcessID theKernelProcess,
                    ItemCount requestedPatchChains,
                    ItemCount *totalPatchChains,
                    PatchChainID *thePatchChains);
```

`theKernelProcess`
The ID of the process with which the patch chains are associated. Specify `kCurrentKernelProcessID` to indicate the current process.

`requestedPatchChains`
An integer specifying the size of the array in which this function stores patch chain information when it returns.

`totalPatchChains`
A pointer to the total number of patch chains associated with the specified kernel process.

`thePatchChains`
A pointer to an array of patch chain IDs. On return, the `GetPatchChainsInKernelProcess` function stores the patch chain IDs in the specified process in this array.

DISCUSSION

The patch chain IDs in the array referenced by `thePatchChains` parameter are not listed in any particular order.

If you call this function and the value specified by the `requestedPatchChains` parameter is smaller than the value specified by the `totalPatchChains` parameter, this means that the array you have allocated for the patch chain IDs is too small. You must set the `requestedPatchChains` parameter to the value specified by the `totalPatchChains` parameter and then call the function again.

SEE ALSO

Use the `GetPatchChainInformation` function (page 1-39) to get information about a specific patch chain.

You use the `GetPatchesInPatchChain` function (page 1-41) to obtain a list of all the patches in a patch chain.

GetPatchChainInformation

Returns information about a patch chain.

```
OSStatus GetPatchChainInformation (PatchChainID thePatchChain,
                    PBVersion version,
                    PatchChainInformation *patchChainInfo);
```

`thePatchChain` The ID of the patch chain about which information is sought.

`version` The constant `kPatchChainInformationVersion`.

`patchChainInfo`
A pointer to a patch chain information structure (page 1-35) that the `GetPatchChainInformation` function fills in when it returns.

DISCUSSION

For a given patch chain ID, the `GetPatchChainInformation` function returns a patch chain information structure that specifies the ID of the process with which the patch chain is associated and the address of the routine that is being patched.

SEE ALSO

Use the `GetPatchChainsInKernelProcess` function (page 1-38) to find out what other patch chains are associated with the process to which this patch chain belongs.

Determining Whether a Routine is a Patch

You use the functions described in this section to determine whether a routine belongs to a patch chain and whether it is a patch or the routine being patched.

GetPatchChainFromProcPtr

Determines whether a routine belongs to a patch chain.

```
OSStatusGetPatchChainFromProcPtr (KernelProcessID theKernelProcess,
                    PatchableProcPtr theRoutine,
                    PatchChainID *thePatchChain);
```

`theKernelProcess`
The process ID of the process to which the patched routine belongs.

`theRoutine` The address of the routine.

`thePatchChain` A pointer to the ID of the patch chain to which the routine belongs either because it is a patch or because it is the routine being patched (the root of the chain).

DISCUSSION

If the parameter `theRoutine` specifies a routine that is either a patch in a patch chain or the root of a patch chain, the `GetPatchChainFromProcPtr` function returns the ID of the patch chain to which the routine belongs. Otherwise, the function returns an error.

SEE ALSO

To determine whether the routine is a patch or a root, use the `GetPatchFromProcPtr` function (page 1-40).

GetPatchFromProcPtr

Determines whether a routine is a patch and returns its ID if it is.

```
OSStatus GetPatchFromProcPtr(KernelProcessID theKernelProcess,
                    PatchableProcPtr theRoutine, PatchID *thePatch);
```

`theKernelProcess`
The ID of the process to which the routine belongs.

`theRoutine` The address of the routine.

`thePatch` A pointer to the ID of the patch. If the routine is a patch, the `GetPatchFromProcPtr` function sets this field when it returns.

DISCUSSION

If the specified routine is a patch, this function references the ID of the patch in the parameter `thePatch`. If the routine is a root or cannot be found in a patch chain, the function returns an error.

SEE ALSO

If the routine is a patch, you can obtain additional information about the patch by calling the `GetPatchInformation` function(page 1-42).

Obtaining Information About a Patch

You use the functions described in this section to obtain a list of all the patches in a patch chain and to obtain information about a particular patch in a chain.

GetPatchesInPatchChain

Returns a list of patches in a patch chain.

```
OSStatus GetPatchesInPatchChain (PatchChainID thePatchChain,
                    ItemCountrequestedPatches, ItemCount * totalPatches,
                    PatchID * thePatches);
```

`thePatchChain` The ID of the patch chain.

`requestedPatches`
An integer specifying the size of the array in which this function stores patch IDs when it returns.

`totalPatches` A pointer to the total number of patch IDs in the patch chain.

`thePatches` A pointer to an array of patch IDs. On return, the `GetPatchesInPatchChain` function stores the patch IDs in the specified patch chain in this array.

DISCUSSION

The patches in the array referenced by the parameter `thePatches` are not listed in any particular order. The function returns the names of all the patches in the patch chain, whether they are enabled or not.

If you call this function and the value specified by the `requestedPatches` parameter is smaller than the value returned in the `totalPatches` parameter, this means that the array you have allocated for the patch ID information is too small. You must set the `requestedPatches` parameter to the value specified by the `totalPatches` parameter and call the function again.

SEE ALSO

Use the `GetPatchInformation` function (page 1-42) to get information about a particular patch in a patch chain.

Use the `GetPatchChainInformation` function (page 1-39) to find out the name of the routine being patched and the ID of the process that contains the patched routine.

GetPatchInformation

Returns information about a patch.

```
OSStatus GetPatchInformation (PatchID thePatchID,PBVersion version,
                    PatchInformation *patchInfo);
```

`thePatchID`	The ID of the patch.
`version`	The value `kPatchInformationVersion`.
`patchInfo`	A pointer to a patch information structure. On return, the `GetPatchInformation` function fills in the fields of this structure.

DISCUSSION

The `GetPatchInformation` function returns the following information about a patch: the ID of the patch chain to which the patch belongs, the address of the patch routine, the name of the patch, the ordering constraints specified for the patch, and any options for the patch. Once a patch has been installed by the

Patch Manager, any client that knows the ID of the patch can obtain patch information by calling the `GetPatchInformation` function.

SEE ALSO

Use the `GetPatchChainsInKernelProcess` function (page 1-38) to obtain the IDs of all patch chains associated with a process. Then you can use the `GetPatchesInChain` function (page 1-41) to obtain the IDs of all patches in a chain.