This chapter describes the Code Fragment Manager, the part of the Macintosh system software that loads fragments into memory and prepares them for execution. A fragment can be an application, an import library, a system extension, or any other block of executable code and its associated data.

The Code Fragment Manager is intended to operate transparently to most applications and other software. You need to use the Code Fragment Manager explicitly only if

- you need to load code modules dynamically during the execution of your application or other software
- you want to unload code modules before the termination of your application
- you want to obtain information about the symbols exported by a fragment

For example, if your application supports dynamic loading of tools, filters, or other software modules contained in fragments, you'll need to use the Code Fragment Manager to load and prepare them for execution.

This chapter also describes the format of the code fragment resource, which defines information about a fragment. You need to create a code fragment resource (a resource of type `'cfrg'`) for each application or import library you create. For information on doing this, see "Creating a Code Fragment Resource" on page 3-12.

To use this chapter, you should already be generally familiar with the Macintosh Operating System. See the books *Inside Macintosh: Processes* and *Inside Macintosh: Memory* for information about the run-time architecture of the 680x0 environment. You also need to be familiar with the run-time architecture of PowerPC processor-based Macintosh computers, as explained in the chapter "Introduction to PowerPC System Software." That chapter describes the general nature and structure of fragments.

This chapter begins by describing the capabilities of the Code Fragment Manager. Then it describes how the Code Fragment Manager searches for the appropriate versions of import libraries. In general, you need to know these details about searching and version checking only if you are creating updated versions of an existing import library. The section "Using the Code Fragment Manager" beginning on page 3-10 provides code samples illustrating how to use some of the routines provided by the Code Fragment Manager. The section "Code Fragment Manager Reference" beginning on page 3-15 is a complete reference to the Code Fragment Manager.

# About the Code Fragment Manager

The Code Fragment Manager is the Operating System loader for executable code and data that are contained in fragments. Its operations are loosely analogous to those of the Segment Manager in previous versions of the Macintosh system software. The Code Fragment Manager, however, provides a much richer set of services than the Segment Manager, including

- loading and preparation of fragments for execution
- automatic resolution of imported symbols by locating and loading import libraries used by a fragment

- automatic execution of a fragment's initialization and termination routines
- support for updated versions of import libraries

The following sections describe how fragments are structured, how the Code Fragment Manager searches fragments for unresolved symbols, and how it manages different versions of import libraries.

## Fragments

The Code Fragment Manager operates primarily on fragments. A **fragment** is a block of executable code and its associated data. Fragments can be loosely differentiated into three categories, based on how they are used:

- applications
- import libraries
- extensions

Fragments contain symbols, some or all of which may be referenced by code or data in other fragments; these kinds of symbols are called **exported symbols** (or, for brevity, **exports**). An import library is a fragment that consists primarily of exported symbols and their associated code and data. Other kinds of fragments can contain references to the exported symbols of an import library; these references are called **imported symbols** (or, for brevity, **imports**).

During the linking phase of building a fragment, the linker creates an import for each external symbol that is resolved to an export from some import library. The code or data referenced by that import is not copied into the fragment. Instead, as part of the process of loading the fragment into memory and preparing it for execution, the Code Fragment Manager replaces the imported symbol with the address of the exported code or data.

**Note**
Both code and data may be exported by name. However, routines are usually exported indirectly, via a transition vector to the routine. A routine's transition vector is stored in the fragment's data area. See "The Table of Contents" on page 1-26 for more details. ◆

A fragment is stored in a **container,** which can be any logically contiguous object accessible by the Operating System. For example, the executable code and global variables of a PowerPC application are typically stored in a fragment in the application's data fork. The Macintosh ROM is itself a container for the import library that exports the Macintosh system software and for several other import libraries. Application extensions, such as dynamically loadable filters or other code modules, can be stored in resources in the application's resource fork. It's better, however, to use the data fork of some file as the container of an application extension fragment. The extension can be put into the application's data fork (either before or after the application's code fragment) or into the data fork of some other file.

**Note**
A single data fork can contain multiple containers. The `'cfrg'` resource in the file's resource fork allows the Operating System to find each individual container in a data fork. ◆

The Code Fragment Manager is responsible for loading fragments (by calling the Code Fragment Loader) and preparing them for execution. It resolves the imported symbols in a fragment, loading and preparing any additional fragments whose exports are referenced by that fragment. Loading a given fragment, such as an application, usually involves loading and preparing additional fragments.

An import library can have its exported symbols imported by any number of other fragments. When the Code Fragment Manager resolves the imports in a particular fragment, it establishes a **connection** to each individual fragment whose code or data that fragment references. In general, the connections are transparent to the importing fragment. If you call the Code Fragment Manager directly, however, it returns a **connection ID** to you that uniquely identifies the connection. You can use the connection ID to perform various actions on the exporting fragment (for example, to break the connection and unload the fragment or to get information about its exported symbols).

**Note**
There is no practical limit on the size of a fragment. ◆

## Import Library Searching

When searching for an import library to find code or data that is imported by some other fragment, the Code Fragment Manager follows a standard search path. It looks in various files and folders in a specific order until it finds an import library that exports the code or data imported by the fragment being loaded. Once the Code Fragment Manager finds a library that it deems compatible with the fragment it's loading, it stops searching and resolves imports in the fragment to code or data in that library. In general, the exact order in which the Code Fragment Manager searches for import libraries is transparent to your software. However, you might need the information in this section to ensure that a particular import library is found before some other import library, which might also be compatible with your fragment.

**Note**
See the next section, "Version Checking" beginning on page 3-7, for information on how the Code Fragment Manager determines whether some import library is compatible with a fragment. ◆

When loading and preparing an application that imports code or data from an import library, the Code Fragment Manager searches first in the application file itself, by looking for import libraries indicated in the application's `'cfrg'` resource. Typically, any import libraries contained in your application are located in your application's data fork, either before or after the container that holds your application's code and data. Less commonly,

you can put an import library into a resource in your application's resource fork. The `'cfrg'` resource specifies the location of any import libraries that you've included with your application, whether in the data or the resource fork.

If an import library used by your application is not found in the application file itself, the Code Fragment Manager next searches in any directory designated as the application's **library directory,** a directory used by the application to store import libraries or aliases to import libraries. You specify a library directory by including in the appropriate field of your `'cfrg'` resource the ID of an alias resource that picks out the library directory. See "The Code Fragment Resource" beginning on page 3-28 for details.

The Code Fragment Manager searches a directory by looking for files of type `'shlb'` that contain a resource of type `'cfrg'`. The `'cfrg'` resource identifies the logical name of the import library, which is needed to match the library's name generated at link time. There can be more than one logical name listed in a single `'cfrg'` resource. This might happen if there are multiple import libraries contained in the data fork of a single `'shlb'` file. This might also happen if a single import library or application is to be identified by more than one name. Within a directory, the Code Fragment Manager also looks for aliases to files of type `'shlb'` and resolves them to their targets. The alias file must itself be of type `'shlb'`.

If no suitable import library has been found yet, the Code Fragment Manager searches next in the directory that contains the application. If any import libraries—whether located in the application's directory or targeted by an alias in the application's directory—are determined to be compatible with the fragment whose imports are being resolved, the Code Fragment Manager chooses the most compatible library and stops searching.

**IMPORTANT**

The Code Fragment Manager looks only in the top level of the application's directory, not in any subdirectories contained in it. ▲

If no suitable import library has been found yet, the Code Fragment Manager searches next in the Extensions folder in the System Folder and in all the subdirectories of the Extensions folder, including any directories that are targets of directory aliases in the Extensions folder. Once again, both files of type `'shlb'` and targets of aliases of type `'shlb'` are candidates for compatibility checking. This scheme allows you to store your import libraries in a vendor-specific location in the Extensions folder.

If the Code Fragment Manager still hasn't found a compatible import library that exports the imported symbols in the fragment it's trying to prepare, it continues by looking in a **ROM registry,** which keeps track of all import libraries that are stored in the ROM of a Macintosh computer. The Code Fragment Manager registers all ROM-based import libraries in this registry at system startup time.

The final stage of the search path is a **file and directory registry** that it maintains internally. This registry is a list of files and directories that, for various reasons, cannot be put into the normal search path followed by the Code Fragment Manager or would not be recognized as import libraries even if they were in that path. For example, to be registered automatically by the Component Manager, a component must be stored in a file of type `'thng'`. To inform the Code Fragment Manager that the file also

contains one or more import libraries in its data fork, it can be registered in the file and directory registry.

**Note**
The Code Fragment Manager routine to register a file or directory is currently private. ◆

If your application or other software loads a fragment explicitly from disk by calling the `GetDiskFragment` routine, the Code Fragment Manager first looks for any needed import libraries in the **load directory,** the directory that contains the fragment being loaded. (This directory is the one specified in the `fileSpec` parameter you pass to `GetDiskFragment`.) If no suitable import library is found there, the search continues along the path followed when loading and preparing an application. However, the Code Fragment Manager looks in the load directory first only if it is different from the application's directory. Otherwise, the load directory is searched in its normal sequence, after the application file itself and the library directory.

In summary, the Code Fragment Manager looks in the following places when searching for an import library to resolve one or more imports in a fragment being loaded:

1. The load directory (the directory containing the fragment being loaded). The load directory, however, is searched only when a fragment is loaded in response to a call to `GetDiskFragment` or `GetSharedLibrary`, and only when it's different from the application's directory.
2. The application file, if the application's `'cfrg'` resource indicates that the application file contains import libraries. The application fragment is implicitly treated here as an import library.
3. The application's library directory (as specified in the application's `'cfrg'` resource).
4. The application's directory. Only the top level of this directory is searched.
5. The Extensions folder in the System Folder. The Extensions folder and all directories in the Extensions folder are searched.
6. The ROM registry maintained internally by the Code Fragment Manager.
7. The file and directory registry maintained internally by the Code Fragment Manager.

At any stage, the Code Fragment Manager selects the one import library of all those available to it that best satisfies its compatibility version checking. If an import library meets the relevant criteria, the library search stops. Otherwise, the search continues to the next stage. If the final stage (the file and directory registry) is reached and no suitable library can be found, the Code Fragment Manager gives up and does not load the original fragment.

## Version Checking

One of the principal benefits of import libraries, aside from their ability to reduce the size of applications and other fragments, is the ease with which a library developer can make improvements in portions of the import library without requiring developers to modify or rebuild any applications that use the import library. The library developer

needs only to ensure that the updated version is compatible with the version expected by the applications using the library. In general, this means that the external programming interface provided by the import library remains unchanged throughout changes in the underlying implementation.

The Code Fragment Manager provides a simple but powerful version-checking scheme intended to prevent incompatibilities between import libraries and the fragments that use them. This checking is always performed automatically as part of the normal fragment loading and preparation process. In general, your application does not need to concern itself with checking the version of an import library whose code or data it uses.

To take a simple example, suppose that an application uses a single import library. When the application is created, it is linked with some version of that library. Unresolved external symbols in the application are resolved, by the linker, to exported code or data in the import library. The version of the import library used at link time is called the **definition version** of the library (because it supplies the definitions of exported symbols, not the actual implementation of routines and initialization of variables).

When the application is loaded and prepared for execution, it must be connected to a version of that import library. The version of the import library used at load time is called the **implementation version** of the library (because it supplies the implementations of routines and initializations of variables exported by the library). The essential requirement is that the implementation version of an import library used at run time be compatible with the definition version used at link time. The two versions do not need to be identical, but they must satisfy the same programming interface. (The implementation can be a superset of the definition library.)

To allow the Code Fragment Manager to check the implementation version of an import library against the definition version used when linking the application, the linker copies version information from the definition library into the application. When the application is launched, the version information in the application is compared with the version information stored in the implementation library. If the version of the import library is identical to that expected by the application, the library and the application are deemed compatible. If, however, the two versions are not identical, the Code Fragment Manager inspects additional information in whichever of the two fragments (the application and the import library) is the newer fragment. The idea is to allow the newer fragment to decide whether it is compatible with the older fragment.

Every import library contains three version numbers: the current version number, the oldest supported definition version number, and the oldest supported implementation version number. The two latter version numbers are included to provide a way for the Code Fragment Manager to determine whether a given definition version is compatible with a given implementation version, if the current versions of the library and the definition version used to link the application are not identical.

**IMPORTANT**

The current version number must always be greater than or equal to both the oldest supported definition version number and the oldest supported implementation version number. ▲

The linker copies into the application both the current version number of the definition library and the oldest supported implementation version number. When the application is launched, the Code Fragment Manager checks those numbers with the version numbers in the implementation libraries according to the algorithm shown in Listing 3-1.

**Listing 3-1** Pseudocode for the version-checking algorithm

```
if (Definition.Current == Implementation.Current)
   return(kLibAndAppAreCompatible);
else if (Definition.Current > Implementation.Current)
   /*definition version is newer than implementation version*/
   if (Definition.OldestImp <= Implementation.Current)
      return(kImplAndDefAreCompatible);
   else
      return(kImplIsTooOld);
else
   /*definition version is older than implementation version*/
   if (Implementation.OldestDef <= Definition.Current)
      return(kImplAndDefAreCompatible);
   else
      return(kDefIsTooOld);
```

If the current version number copied into the application from the definition library at link time is the same as the current version number of the candidate version of the implementation import library, then the Code Fragment Manager accepts that version of the implementation import library and continues with the loading and preparation of the application. Otherwise, the Code Fragment Manager determines which of the two fragments is newer and then applies a further check.

If the current version number copied into the application from the definition library at link time is greater than the current version number of the candidate version of the implementation import library, the Code Fragment Manager compares the oldest supported implementation version number in the application with the current version number of the implementation library. If the definition library's oldest supported implementation version number is less than or equal to the library's current version number, the application and library are deemed compatible. Otherwise, the library is too old for the application.

If the current version number copied into the application from the definition library at link time is less than the current version number of the most recent version of the implementation import library, the Code Fragment Manager compares the oldest supported definition library version number (stored in the implementation library) with the current definition library version number (stored in the application). If the oldest supported definition library version number is less than or equal to the application's current version number, the application and library are deemed compatible. Otherwise, the application is too old for the library.

**Note**
In general, of course, the Code Fragment Manager checks the compatibility of a fragment being loaded and *all* of the import libraries from which it imports code and data. ◆

The version numbers in both the definition and implementation versions of an import library should have the same format as the first 4 bytes of a version resource (that is, a resource of type `'vers'`). See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for complete information on version resources. When comparing version numbers, however, the Code Fragment Manager treats those 4 bytes simply as an unsigned long quantity. As a result, the value 0x00000000 is interpreted as a valid version number.

# Using the Code Fragment Manager

The Code Fragment Manager provides routines that you can use to explicitly load code fragments and to get information about symbols exported by a particular fragment. This section illustrates how to use those routines.

**IMPORTANT**
In general, the Code Fragment Manager automatically loads all import libraries required by your application at the time your application is launched. You need to use the routines described in this section only if your application supports dynamically loaded application tools, filters, or other code modules. ▲

This section also describes how to create a code fragment resource. Every application and import library must have a code fragment resource to describe basic information about the application or import library.

## Loading Code Fragments

You can use the Code Fragment Manager to load fragments from the containers in which they are stored. You need to do this only for code fragments that are dynamically added to your application's context during execution. This might happen, for instance, if your application supports dynamically loadable filters or tools.

The executable code you want to bind to your application context can be stored in any kind of container. If the container is an import library (a file of type `'shlb'`), you can use the Code Fragment Manager's `GetSharedLibrary` function. If the container is a disk file, you call the `GetDiskFragment` function. If the container is a resource, you need to load the resource into memory (using normal Resource Manager routines) and then call the `GetMemFragment` function. See "Loading Fragments" beginning on page 3-19 for complete details on each of these functions.

Listing 3-2 and Listing 3-3 illustrate how to load application-specific tools into memory using the Code Fragment Manager. Listing 3-2 shows how to load a resource-based fragment.

**Listing 3-2** Loading a resource-based fragment

```
Handle         myHandle;
OSErr          myErr;
ConnectionID   myConnID;
Ptr            myMainAddr;
Str255         myErrName;

myHandle = GetResource('tool', 128);
HLock(myHandle);
myErr = GetMemFragment(*myHandle, GetHandleSize(myHandle),
               myToolName, kLoadNewCopy, &myConnID,
               (Ptr*)&myMainAddr, myErrName);
if (myErr) {
   AlertUser(myErr);
   goto noLoad;
}
```

As you can see, Listing 3-2 loads the resource into memory by calling the Resource Manager function `GetResource` and locks it by calling the Memory Manager procedure `HLock`. Then it calls `GetMemFragment` to prepare the fragment. The first parameter passed to `GetMemFragment` specifies the address in memory of the fragment. Because `GetResource` returns a handle to the resource data, Listing 3-2 dereferences the handle to obtain a pointer to the resource data. To avoid dangling pointers, you need to lock the block of memory before calling `GetMemFragment`. The constant `kLoadNewCopy` passed as the fourth parameter requests that the Code Fragment Manager allocate a new copy of the fragment's global data section.

Listing 3-3 shows how to load a disk-based fragment.

**Listing 3-3** Loading a disk-based fragment

```
myErr = GetDiskFragment(&myFSSpec, 0, kWholeFork, myToolName,
               kLoadNewCopy, &myConnID, (Ptr*)&myMainAddr,
               myErrName);
if (myErr) {
   AlertUser(myErr);
   goto noLoad;
}
```

All import libraries and other fragments that are loaded on behalf of your application (either as part of its normal startup or programmatically by your application) are unloaded by the Process Manager at application termination; therefore, a library can be loaded and does not have to be unloaded by the application before it terminates.

## Creating a Code Fragment Resource

You need to create a **code fragment resource** (a resource of type `'cfrg'`) for each native application or import library you create. This resource identifies the instruction set architecture, location, size, and logical name of the application or import library, as well as version information for import libraries.

In PowerPC or fat applications, the code fragment resource is read by the Process Manager at application launch time. The Process Manager needs to know whether the application contains PowerPC code and, if so, where that code is located. If the Process Manager cannot find a `'cfrg'` resource in the application's resource fork, it assumes that the application is a 680x0 application, where the executable code is contained within `'CODE'` resources in the application's resource fork.

**IMPORTANT**

A code fragment resource must have resource ID 0. ▲

For an application, the code fragment resource typically indicates that the application's executable code fragment begins at offset 0 within the application's data fork and extends for the entire length of the data fork. Listing 3-4 shows the Rez input for a typical application's code fragment resource.

**Listing 3-4** The Rez input for a typical application's `'cfrg'` resource

```
#include "CodeFragmentTypes.r"
resource 'cfrg' (0) {
   {
      kPowerPC,              /*instruction set architecture*/
      kFullLib,              /*no update level for apps*/
      kNoVersionNum,         /*no current version number*/
      kNoVersionNum,         /*no oldest def'n version number*/
      kDefaultStackSize,     /*use default stack size*/
      kNoAppSubFolder,       /*no library directory*/
      kIsApp,                /*fragment is an application*/
      kOnDiskFlat,           /*fragment is on disk*/
      kZeroOffset,           /*fragment starts at fork start*/
      kWholeFork,            /*fragment occupies entire fork*/
      "SurfWriter"           /*name of the application*/
   }
};
```

**Note**
See "The Code Fragment Resource" on page 3-28 for complete information about the structure of a code fragment resource. ◆

For import libraries, the code fragment resource is read by the Code Fragment Manager as part of the process of searching for symbols imported by some fragment that is currently being loaded and prepared for execution. (See the section "Import Library Searching" on page 3-5 for details on how the Code Fragment Manager searches for import libraries.) The information in the `'cfrg'` resource is also used to ensure that the Code Fragment Manager finds an implementation version of an import library that is compatible with the definition version used to link the fragment being loaded and prepared for execution. Listing 3-4 shows the Rez input for a typical code fragment resource for an import library.

**Listing 3-5** The Rez input for a typical import library's `'cfrg'` resource

```
#define kOldDefVers     0x01008000     /*version 1.0*/
#define kCurrVers       0x02008000     /*version 2.0*/

#include "CodeFragmentTypes.r"
resource 'cfrg' (0) {
   {
      kPowerPC,              /*instruction set architecture*/
      kFullLib,              /*base library*/
      kCurrVers,             /*current version number*/
      kOldDefVers,           /*oldest definition version number*/
      kDefaultStackSize,     /*ignored for import library*/
      kNoAppSubFolder,       /*ignored for import library*/
      kIsLib,                /*fragment is a library*/
      kOnDiskFlat,           /*fragment is on disk*/
      kZeroOffset,           /*fragment starts at fork start*/
      kWholeFork,            /*fragment occupies entire fork*/
      "SurfTools"            /*name of the library*/
   }
};
```

An import library's code fragment resource also specifies the logical name of the import library. This is the name used by the Code Fragment Manager to resolve imports in some other fragment. The logical name can be different from the name of the file containing the import library.

Note that code fragment resources are required only for fragments that are either applications or import libraries. If you need similar version-checking or name-binding capabilities for fragments that are application extensions, you will need to provide your own code to do this.

## Getting Information About Exported Symbols

In cases in which you load a fragment programmatically (that is, by calling Code Fragment Manager routines), you can get information about the symbols exported by that fragment by calling the `CountSymbols` and `GetIndSymbol` functions. The `CountSymbols` function returns the total number of symbols exported by a fragment. `CountSymbols` takes as one of its parameters a connection ID; accordingly, you must already have established a connection to a fragment before you can determine how many symbols it exports.

Given an index ranging from 1 to the total number of symbols in a fragment, the `GetIndSymbol` function returns the name, address, and class of a symbol in that fragment. You can use `CountSymbols` in combination with `GetIndSymbol` to get information about all the symbols in a fragment. For example, the code in Listing 3-6 prints the names of all the symbols in a particular fragment.

**Listing 3-6** Finding symbol names

```
void MyGetSymbolNames (ConnectionID myConnID);
{
   long           myIndex;
   long           myCount;       /*number of exported symbols in fragment*/
   OSErr          myErr;
   Str255         myName;        /*symbol name*/
   Ptr            myAddr;        /*symbol address*/
   SymClass       myClass;       /*symbol class*/

   myErr = CountSymbols(myConnID, &myCount);
   if (!myErr)
      for (myIndex = 1; myIndex <= myCount; myIndex++)
         {
            myErr = GetIndSymbol(myConnID, myIndex, myName,
                                 &myAddr, &myClass);
            if (!myErr)
               printf("%P", myName);
         }
}
```

If you already know the name of a particular symbol whose address and class you want to determine, you can use the `FindSymbol` function. See page 3-24 for details on calling `FindSymbol`.

# Code Fragment Manager Reference

This section describes the data structures and routines provided by the Code Fragment Manager. See "Using the Code Fragment Manager" beginning on page 3-10 for detailed instructions on using these routines. This section also describes the format of the optional initialization and termination routines you can include in a fragment, as well as the structure of the code fragment resource.

## Data Structures

This section describes the data structures that define the format of the data passed to a fragment's initialization routine.

**IMPORTANT**

You need the information in this section only if your fragment (application, import library, or extension) contains an initialization routine. In addition, much of the information passed to an initialization routine is intended for use by language implementors. Most other developers are likely to need only the pointer to a file specification record passed to disk-based fragments. (This information allows the initialization routine to access its own resource fork.) ▲



### Fragment Initialization Block

The Code Fragment Manager passes to your fragment's initialization routine a pointer to a **fragment initialization block,** which contains information about the fragment. A fragment initialization block is defined by the `InitBlock` data type.

```
struct InitBlock {
   long                  contextID;     /*context ID*/
   long                  closureID;     /*closure ID*/
   long                  connectionID;  /*connection ID*/
   FragmentLocator       fragLocator;   /*fragment location*/
   Ptr                   libName;       /*pointer to fragment name*/
   long                  reserved4a;    /*reserved*/
   long                  reserved4b;    /*reserved*/
   long                  reserved4c;    /*reserved*/
   long                  reserved4d;    /*reserved*/
};
typedef struct InitBlock InitBlock, *InitBlockPtr;
```

**Field descriptions**

| | |
|---|---|
| `contextID` | A context ID. |
| `closureID` | A closure ID. |
| `connectionID` | A connection ID. |
| `fragLocator` | A fragment location record that specifies the location of the fragment. See the following section for details about the structure of a fragment location record. |
| `libName` | A pointer to the name of the fragment being initialized. The name is a Pascal string (a length byte followed by the name itself). |
| `reserved4a` | Reserved for use by Apple Computer. |
| `reserved4b` | Reserved for use by Apple Computer. |
| `reserved4c` | Reserved for use by Apple Computer. |
| `reserved4d` | Reserved for use by Apple Computer. |

**IMPORTANT**

The fields of a fragment initialization block are aligned in memory in accordance with 680x0 alignment conventions. ▲

## Fragment Location Record

The `fragLocator` field of an initialization block contains a **fragment location record** that provides information about the location of a fragment. A fragment location record is defined by the `FragmentLocator` data type.

```
struct FragmentLocator {
   long                    where;         /*location selector*/
   union {
      MemFragment          inMem;         /*memory location record*/
      DiskFragment         onDisk;        /*disk location record*/
      SegmentedFragment    inSegs;        /*segment location record*/
   } u;
};
typedef struct FragmentLocator FragmentLocator, *FragmentLocatorPtr;
```

**Field descriptions**

`where` A selector that determines which member of the following union is relevant. This field can contain one of these constants:

```
enum {
   kInMem,           /*container in memory*/
   kOnDiskFlat,      /*container in a data fork*/
   kOnDiskSegmented  /*container in a resource*/
};
```

| | |
|---|---|
| `inMem` | A memory location record. |
| `onDisk` | A disk location record. |
| `inSegs` | A segment location record. |

**IMPORTANT**
The fields of a fragment location record are aligned in memory in accordance with 680x0 alignment conventions. ▲

## Memory Location Record

For fragments located in memory, the `inMem` field of a fragment location record contains a **memory location record,** which specifies the location of the fragment in memory. A memory location record is defined by the `MemFragment` data type.

```
struct MemFragment {
   Ptr                     address;       /*pointer to start of fragment*/
   long                    length;        /*length of fragment*/
   Boolean                 inPlace;       /*is data section in place?*/
};
typedef struct MemFragment MemFragment;
```

**Field descriptions**

`address` A pointer to the beginning of the fragment in memory.

`length` The length, in bytes, of the fragment.

`inPlace` A Boolean value that specifies whether the container's data section is instantiated in place (`true`) or elsewhere (`false`).

**IMPORTANT**
The fields of a memory location record are aligned in memory in accordance with 680x0 alignment conventions. ▲

## Disk Location Record

For fragments located in the data fork of a file on disk, the `onDisk` field of a fragment location record contains a **disk location record,** which specifies the location of the fragment. A disk location record is defined by the `DiskFragment` data type.

```
struct DiskFragment {
   FSSpecPtr               fileSpec;      /*pointer to FSSpec*/
   long                    offset;        /*offset to start of fragment*/
   long                    length;        /*length of fragment*/
};
typedef struct DiskFragment DiskFragment;
```

**Field descriptions**

`fileSpec` A pointer to a file specification record (a data structure of type `FSSpec`) for the data fork of a file. This pointer is valid only while the initialization routine is executing. If you need to access the information in the file specification record at any later time, you must make a copy of that record.

| | |
|---|---|
| `offset` | The offset, in bytes, from the beginning of the file's data fork to the beginning of the fragment. |
| `length` | The length, in bytes, of the fragment. If this field contains the value 0, the fragment extends to the end-of-file. |

**IMPORTANT**

The fields of a disk location record are aligned in memory in accordance with 680x0 alignment conventions. ▲

## Segment Location Record

For fragments located in the resource fork of a file on disk, the `inSegs` field of a fragment location record contains a **segment location record,** which specifies the location of the fragment. A segment location record is defined by the `SegmentedFragment` data type.

```
struct SegmentedFragment {
   FSSpecPtr               fileSpec;      /*pointer to FSSpec*/
   OSType                  rsrcType;      /*resource type*/
   short                   rsrcID;        /*resource ID*/
};
typedef struct SegmentedFragment SegmentedFragment;
```

**Field descriptions**

| | |
|---|---|
| `fileSpec` | A pointer to a file specification record (a data structure of type `FSSpec`) for the resource fork of a file. This pointer is valid only while the initialization routine is executing. If you need to access the information in the file specification record at any later time, you must make a copy of that record. |
| `rsrcType` | The resource type of the resource containing the fragment. |
| `rsrcID` | The resource ID of the resource containing the fragment. |

**IMPORTANT**

The fields of a segment location record are aligned in memory in accordance with 680x0 alignment conventions. ▲

# Code Fragment Manager Routines

You can use the routines provided by the Code Fragment Manager to

- load a fragment by filename or library name
- identify an import library that is already loaded
- unload a fragment
- find a symbol by name in a fragment
- find all the symbols in a fragment

## Loading Fragments

The Code Fragment Manager provides three functions that you can use to load various kinds of fragments: `GetDiskFragment`, `GetMemFragment`, and `GetSharedLibrary`. Loading involves finding the specified fragment, reading it into memory (if it isn't already in memory), and preparing it for execution. The Code Fragment Manager attempts to resolve all symbols imported by the fragment; to do so may involve loading import libraries.

If the fragment loading fails, the Code Fragment Manager returns an error code. Note, however, that the error encountered is not always in the fragment you asked to load. Rather, the error might have occurred while attempting to load an import library that the fragment you want to load depends on. For this reason, the Code Fragment Manager also returns, in the `errName` parameter, the name of the fragment that caused the load to fail. Although fragment names are restricted to 63 characters, the `errName` parameter is declared as type `Str255`; doing this allows future versions of the Code Fragment Manager to return a more informative message in the `errName` parameter.



## GetDiskFragment

You can use the `GetDiskFragment` function to locate and possibly also load a fragment contained in a file's data fork into your application's context.

```
OSErr GetDiskFragment (FSSpecPtr fileSpec, long offset,
                       long length, Str63 fragName,
                       LoadFlags findFlags, ConnectionID *connID,
                       Ptr *mainAddr, Str255 errName);
```

`fileSpec` A file system specification that identifies the disk-based fragment to load.

`offset` The number of bytes from the beginning of the file's data fork at which the beginning of the fragment is located.

`length` The length (in bytes) of the fragment. Specify the constant `kWholeFork` for this parameter if the fragment extends to the end-of-file of the data fork. Specify a nonzero value for the exact length of the fragment.

`fragName` An optional name of the fragment. (This information is used primarily to allow you to identify the fragment during debugging.)

`findFlags` A flag that specifies the operation to perform on the fragment. See the description below for the values you can pass in this parameter.

`connID` On exit, the connection ID that identifies the connection to the fragment. You can pass this ID to other Code Fragment Manager routines.

`mainAddr` On exit, the main address of the fragment. The value returned is specific to the fragment itself. Your application can use this parameter for its own purposes.

`errName` On exit, the name of the fragment that could not successfully be loaded. This parameter is meaningful only if the call to `GetDiskFragment` fails.

### DESCRIPTION

The `GetDiskFragment` function locates and possibly also loads a disk-based fragment into your application's context. The actions of `GetDiskFragment` depend on the action flag you pass in the `findFlags` parameter. The Code Fragment Manager recognizes these constants:

```
enum {
   kLoadLib          = 1,  /*load fragment*/
   kFindLib          = 2,  /*find fragment*/
   kLoadNewCopy      = 5   /*load fragment with new copy of data*/
};
```

The `kFindLib` constant specifies that the Code Fragment Manager search for the specified fragment. If the fragment is already prepared and connected to your application, `GetDiskFragment` returns `fragNoErr` as its function result and the existing connection ID in the `connID` parameter. If the specified fragment is not found, `GetDiskFragment` returns the result code `fragLibNotFound`. If the specified fragment is found but could not be connected to your application, `GetDiskFragment` returns the result code `fragLibConnErr`.

The `kLoadLib` constant specifies that the Code Fragment Manager search for the specified fragment and, if it finds it, load it into memory. If the fragment has already been loaded, it's not loaded again. The Code Fragment Manager uses the data-instantiation method specified in the fragment's container (which is either global or per-connection instantiation).

The `kLoadNewCopy` constant specifies that the Code Fragment Manager load the specified fragment, creating a new copy of any writable data maintained by the fragment. You specify `kLoadNewCopy` to obtain one instance per load of the fragment's data and to override the data-instantiation method specified in the container itself. This is most useful for application extensions (for example, drop-in tools).

### RESULT CODES

| | | |
|---|---|---|
| `fragNoErr` | 0 | No error |
| `paramErr` | –50 | Parameter error |
| `fragLibNotFound` | –2804 | Specified fragment not found |
| `fragHadUnresolveds` | –2807 | Loaded fragment has unacceptable unresolved symbols |
| `fragNoMem` | –2809 | Not enough memory for internal bookkeeping |
| `fragNoAddrSpace` | –2810 | Not enough memory in user's address space for section |
| `fragObjectInitSeqErr` | –2812 | Order error during user initialization function |
| `fragImportTooOld` | –2813 | Import library is too old |
| `fragImportTooNew` | –2814 | Import library is too new |
| `fragInitLoop` | –2815 | Circularity in required initialization order |
| `fragLibConnErr` | –2817 | Error connecting to fragment |
| `fragUserInitProcErr` | –2821 | Initialization procedure did not return `noErr` |

**SEE ALSO**

See "Loading Code Fragments" on page 3-10 for more details on the fragment-loading process.

## GetMemFragment

You can use the `GetMemFragment` function to prepare a memory-based fragment.

```
OSErr GetMemFragment (Ptr memAddr, long length, Str63 fragName,
                      LoadFlags findFlags, ConnectionID *connID,
                      Ptr *mainAddr, Str255 errName);
```

| | |
|---|---|
| `memAddr` | The address of the fragment. |
| `length` | The size, in bytes, of the fragment. |
| `fragName` | The name of the fragment. (This information is used primarily to allow you to identify the fragment during debugging.) |
| `findFlags` | A flag that specifies the operation to perform on the fragment. See the description of the `GetDiskFragment` function on page 3-19 for the values you can pass in this parameter. |
| `connID` | On exit, the connection ID that identifies the connection to the fragment. You can pass this ID to other Code Fragment Manager routines (for example, `CloseConnection`). |
| `mainAddr` | On exit, the main address of the fragment. The value returned is specific to the fragment itself. |
| `errName` | On exit, the name of the fragment that could not successfully be loaded. This parameter is meaningful only if the call to `GetMemFragment` fails. |

**DESCRIPTION**

The `GetMemFragment` function prepares for subsequent execution a fragment that is already loaded into memory. This function is most useful for handling code that is contained in a resource. You can read the resource data into memory using normal Resource Manager routines (for example, `Get1Resource`) and then call `GetMemFragment` to complete the processing required to prepare it for use (for example, to resolve any imports and execute the fragment's initialization routine).

▲ **WARNING**

You must lock the resource-based fragment into memory (for example, by calling `HLock`) before calling `GetMemFragment`. You must not unlock the memory until you've closed the connection to the fragment (by calling `CloseConnection`). ▲

**RESULT CODES**

| | | |
|---|---|---|
| `fragNoErr` | 0 | No error |
| `paramErr` | –50 | Parameter error |
| `fragLibNotFound` | –2804 | Specified fragment not found |
| `fragHadUnresolveds` | –2807 | Loaded fragment has unacceptable unresolved symbols |
| `fragNoMem` | –2809 | Not enough memory for internal bookkeeping |
| `fragNoAddrSpace` | –2810 | Not enough memory in user's address space for section |
| `fragObjectInitSeqErr` | –2812 | Order error during user initialization function |
| `fragImportTooOld` | –2813 | Import library is too old |
| `fragImportTooNew` | –2814 | Import library is too new |
| `fragInitLoop` | –2815 | Circularity in required initialization order |
| `fragLibConnErr` | –2817 | Error connecting to fragment |
| `fragUserInitProcErr` | –2821 | Initialization procedure did not return `noErr` |

**SEE ALSO**

See "Loading Code Fragments" on page 3-10 for more details on the fragment-loading process.

## GetSharedLibrary

You can use the `GetSharedLibrary` function to locate and possibly also load an import library into your application's context.

```
OSErr GetSharedLibrary (Str63 libName, OSType archType,
                        LoadFlags findFlags,
                        ConnectionID *connID, Ptr *mainAddr,
                        Str255 errName);
```

`libName` The name of an import library.

`archType` The instruction set architecture of the import library. For the PowerPC architecture, use the constant `kPowerPCArch`. For the 680x0 architecture, use the constant `kMotorola68KArch`.

`findFlags` A flag that specifies the operation to perform on the import library. See the description of the `GetDiskFragment` function on page 3-19 for the values you can pass in this parameter.

`connID` On exit, the connection ID that identifies the connection to the import library. You can pass this ID to other Code Fragment Manager routines.

`mainAddr` On exit, the main address of the import library. The value returned is specific to the import library itself and is not used by the Code Fragment Manager.

`errName` On exit, the name of the fragment that could not successfully be loaded. This parameter is meaningful only if the call to `GetSharedLibrary` fails.

**DESCRIPTION**

The `GetSharedLibrary` function locates the import library named by the `libName` parameter and possibly also loads that import library into your application's context. The actions of `GetSharedLibrary` depend on the action flag you pass in the `findFlags` parameter; pass `kFindLib` to get the connection ID of an existing connection to the specified fragment, `kLoadLib` to load the specified fragment, or `kLoadNewCopy` to load the fragment with a new copy of the fragment's data section.

The `GetSharedLibrary` function does not resolve any unresolved imports in your application. In particular, you cannot use it to resolve any weak imports in your code fragment.

**RESULT CODES**

| | | |
|---|---|---|
| `fragNoErr` | 0 | No error |
| `paramErr` | –50 | Parameter error |
| `fragLibNotFound` | –2804 | Specified fragment not found |
| `fragHadUnresolveds` | –2807 | Loaded fragment has unacceptable unresolved symbols |
| `fragNoMem` | –2809 | Not enough memory for internal bookkeeping |
| `fragNoAddrSpace` | –2810 | Not enough memory in user's address space for section |
| `fragObjectInitSeqErr` | –2812 | Order error during user initialization function |
| `fragImportTooOld` | –2813 | Import library is too old |
| `fragImportTooNew` | –2814 | Import library is too new |
| `fragInitLoop` | –2815 | Circularity in required initialization order |
| `fragLibConnErr` | –2817 | Error connecting to fragment |
| `fragUserInitProcErr` | –2821 | Initialization procedure did not return `noErr` |

**SEE ALSO**

See "Loading Code Fragments" on page 3-10 for more details on the fragment-loading process.

## Unloading Fragments

The Code Fragment Manager provides one function that you can use to close an existing connection to a fragment.

### CloseConnection

You can use the `CloseConnection` function to close a connection to a fragment.

```
OSErr CloseConnection (ConnectionID *connID);
```

`connID` A connection ID.

#### DESCRIPTION

The `CloseConnection` function closes the connection to a fragment indicated by the `connID` parameter. `CloseConnection` decrements the count of existing connections to the specified fragment and, if the resulting count is 0, calls the fragment's termination routine and releases the memory occupied by the code and data sections of the fragment. If the resulting count is not 0, any per-connection data is released but the code section remains in memory.

When a fragment is unloaded as a result of its final connection having been closed, all libraries that depend on that fragment are also released, provided that their usage counts are also 0.

The Code Fragment Manager automatically closes any connections that remain open at the time `ExitToShell` is called for your application, so you need to call `CloseConnection` only for fragments you wish to unload before your application terminates.

#### SPECIAL CONSIDERATIONS

You can close a connection only to the root of a loading sequence (that is, the fragment whose loading triggered the entire load chain).

#### RESULT CODES

| | | |
|---|---|---|
| `fragNoErr` | 0 | No error |
| `fragConnectionIDNotFound` | –2801 | Connection ID is not valid |

## Finding Symbols

The Code Fragment Manager provides three functions that you can use to find the symbols exported by a fragment and get information about them: `FindSymbol`, `CountSymbols`, and `GetIndSymbol`.

## FindSymbol

You can use the `FindSymbol` function to search for a specific exported symbol.

```
OSErr FindSymbol (ConnectionID connID, Str255 symName,
                  Ptr *symAddr, SymClass *symClass);
```

| | |
|---|---|
| `connID` | A connection ID. |
| `symName` | A symbol name. |
| `symAddr` | On exit, the address of the symbol whose name is `symName`. |
| `symClass` | On exit, the class of the symbol whose name is `symName`. See the description below for a list of the recognized symbol classes. |

**DESCRIPTION**

The `FindSymbol` function searches the code fragment identified by the `connID` parameter for the symbol whose name is specified by the `symName` parameter. If that symbol is found, `FindSymbol` returns the address of the symbol in the `symAddr` parameter and the class of the symbol in the `symClass` parameter. The currently recognized symbol classes are defined by constants.

```
enum {
   kCodeSymbol       = 0,  /*a code symbol*/
   kDataSymbol       = 1,  /*a data symbol*/
   kTVectSymbol      = 2   /*a transition vector symbol*/
};
```

Because a fragment's code is normally exported through transition vectors to that code, the value `kCodeSymbol` is not returned in the PowerPC environment. You can use the other two constants to distinguish exports that represent code (of class `kTVectSymbol`) from those that represent general data (of class `kDataSymbol`).

**RESULT CODES**

| | | |
|---|---|---|
| `fragNoErr` | 0 | No error |
| `fragConnectionIDNotFound` | –2801 | Connection ID is not valid |
| `fragSymbolNotFound` | –2802 | Symbol was not found in connection |

## CountSymbols

You can use the `CountSymbols` function to determine how many symbols are exported from a specified fragment.

```
OSErr CountSymbols (ConnectionID connID, long *symCount);
```

`connID` A connection ID.

`symCount` On exit, the number of exported symbols in the fragment whose connection ID is `connID`.

**DESCRIPTION**

The `CountSymbols` function returns, in the `symCount` parameter, the number of symbols exported by the fragment whose connection ID is `connID`. You can use the value returned in `symCount` to index through all the exported symbols in a particular fragment (using the `GetIndSymbol` function).

**RESULT CODES**

| | | |
|---|---|---|
| `fragNoErr` | 0 | No error |
| `fragConnectionIDNotFound` | –2801 | Connection ID is not valid |

## GetIndSymbol

You can use the `GetIndSymbol` function to get information about the exported symbols in a fragment.

```
OSErr GetIndSymbol (ConnectionID connID, long symIndex,
                    Str255 symName, Ptr *symAddr,
                    SymClass *symClass);
```

| | |
|---|---|
| `connID` | A connection ID. |
| `symIndex` | A symbol index. The value of this parameter should be greater than or equal to 1 and less than or equal to the value returned by the `CountSymbols` function. |
| `symName` | On exit, the name of the indicated symbol. |
| `symAddr` | On exit, the address of the indicated symbol. |
| `symClass` | On exit, the class of the indicated symbol. |

**DESCRIPTION**

The `GetIndSymbol` function returns information about a particular symbol exported by the fragment whose connection ID is `connID`. If `GetIndSymbol` executes successfully, it returns the symbol's name, starting address, and class in the `symName`, `symAddr`, and `symClass` parameters, respectively. See the description of the `FindSymbol` function (page 3-24) for a list of the values that can be returned in the `symClass` parameter.

A fragment's exported symbols are retrieved in no predetermined order.

**RESULT CODES**

| | | |
|---|---|---|
| `fragNoErr` | 0 | No error |
| `fragConnectionIDNotFound` | –2801 | Connection ID is not valid |
| `fragSymbolNotFound` | –2802 | Symbol was not found in connection |

# Fragment-Defined Routines

This section describes the initialization and termination routines that you can define for a fragment.

## ConnectionInitializationRoutine

You can define a fragment initialization routine that is executed by the Code Fragment Manager when the fragment is first loaded into memory and prepared for execution. An initialization routine has the following type definition:

```
typedef OSErr ConnectionInitializationRoutine
                                        (InitBlockPtr initBlkPtr);
```

`initBlkPtr`
A pointer to a fragment initialization block specifying information about the fragment.

**Parameter block**

| | | | |
|---|---|---|---|
| → | `contextID` | `long` | A context ID. |
| → | `closureID` | `long` | A closure ID. |
| → | `connectionID` | `long` | A connection ID. |
| → | `fragLocator` | `FragmentLocator` | A fragment location block. |
| → | `libName` | `Ptr` | A pointer to fragment's name. |
| → | `reserved4a` | `long` | Reserved. |
| → | `reserved4b` | `long` | Reserved. |
| → | `reserved4c` | `long` | Reserved. |
| → | `reserved4d` | `long` | Reserved. |

**DESCRIPTION**

A fragment's initialization routine is executed immediately after the fragment has been loaded into memory (if necessary) and prepared for execution, and immediately before the fragment's main routine (if it has one) is executed. The initialization routine is passed a pointer to an initialization block, which contains information about the fragment, such as its location and connection ID. See "Fragment Initialization Block" on page 3-15 for a description of the fields of the initialization block.

You can use the initialization routine to perform any tasks that need to be performed before any of the code or data in the fragment is accessed. For example, you might want to open the fragment's resource fork (if it has one). You can determine the location of the fragment's container from the `FragmentLocator` field of the fragment initialization block whose address is passed to your initialization routine.

**RESULT CODES**

Your initialization routine should return `noErr` if it executes successfully, and some other result code if it does not. If your initialization routine returns any result code other than `noErr`, the entire load fails and the error `fragUserInitProcErr` is returned to the code that requested the root load.

### ConnectionTerminationRoutine

You can define a fragment termination routine that is executed by the Code Fragment Manager when a fragment is unloaded from memory. A termination routine has the following type definition:

```
typedef void ConnectionTerminationRoutine (void);
```

**DESCRIPTION**

A fragment's termination routine is executed immediately before the fragment is unloaded from memory. You can use the termination routine to perform any necessary clean-up tasks, such as closing open resource files or disposing of any memory allocated by the fragment.

Note that a termination routine is not passed any parameters and does not return any result. You are expected to maintain any information about the fragment (such as file reference numbers of any open files) in its static data area.

## Resources

This section describes the code fragment resource, a resource of type `'cfrg'` that is used by the Code Fragment Manager when loading fragments such as applications and import libraries.

This section describes the structure of this resource after it is compiled by the Rez resource compiler, available from APDA. If you are interested in creating the Rez input file for this resource, see "Creating a Code Fragment Resource" on page 3-12 for detailed information.

### The Code Fragment Resource

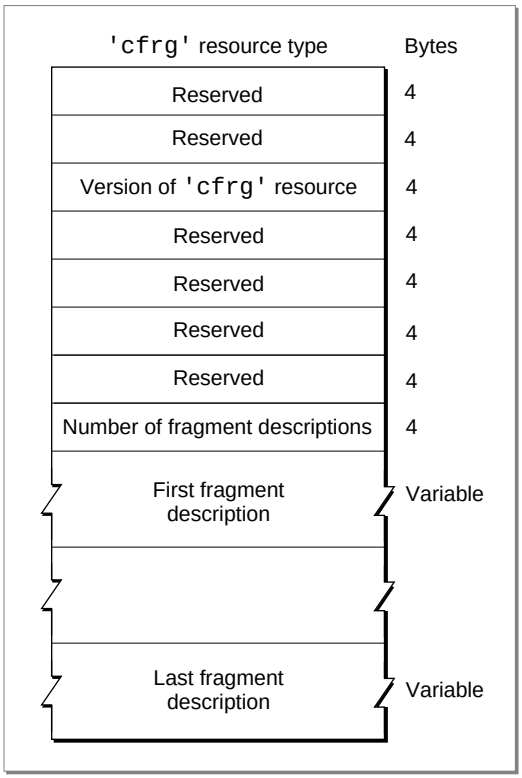
You use a code fragment resource to specify some characteristics of a code fragment. For an application, the code fragment resource indicates to the Process Manager that the application's data fork contains an executable code fragment. For an import library, the code fragment resource specifies the library's name and version information.

**IMPORTANT**

A code fragment resource must have resource ID 0. ▲

Figure 3-1 shows the structure of a compiled code fragment resource.

**Figure 3-1** Structure of a compiled code fragment (`'cfrg'`) resource

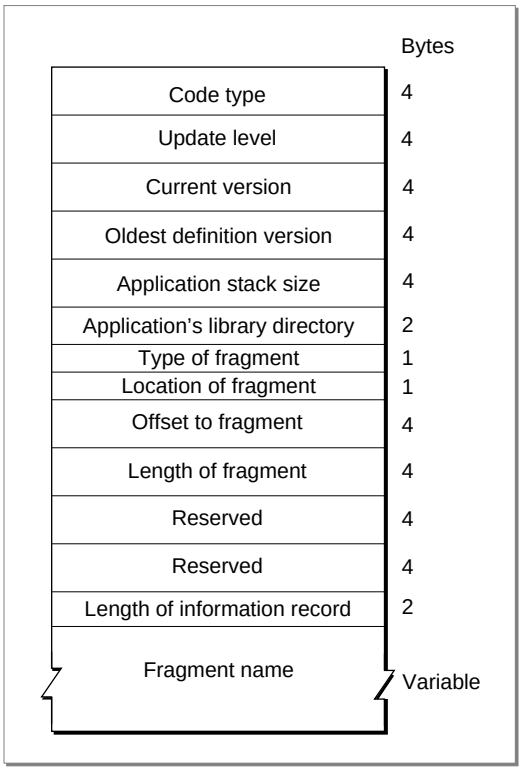



The compiled version of a code fragment resource contains the following elements:

- Reserved. The first two long integers are reserved and should be set to 0.
- Version information. This field specifies the current version of the `'cfrg'` resource. The current version is 0x00000001.
- Reserved. The next four long integers are reserved and should be set to 0.
- Number of fragment descriptions. This field specifies the number of code fragment information records that follow this field in the resource. (The value in this field should be the actual number of information records that follow, beginning with 1.)

Following the array count is an array of **code fragment information records.** A single file can include one or more containers. Similarly, it might occasionally be useful to assign more than one name to a single import library or application. Typically, however, both applications and import libraries include just a single code fragment information record in their `'cfrg'` resources. Each record has the format illustrated in Figure 3-2.

**Figure 3-2** The format of a code fragment information record




A code fragment information record contains the following elements:

- The instruction set architecture. You can use the Rez constant `kPowerPC` (`'pwpc'`) to specify the PowerPC instruction set architecture.
- The update level. For an import library, you can specify either the value `kFullLib` (0), to indicate that the library is a base library (not an update of some other library), or the value `kUpdateLib` (1), to indicate that the library updates only part of some other library. Applications should specify the value `kFullLib` in this field.
- The current version number. For an import library, this field specifies the implementation version. This field has the same format as the first 4 bytes of a resource of type `'vers'`. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for details on the structure of a `'vers'` resource.
- The oldest definition version number. For an import library, this field specifies the oldest version of the definition library with which the implementation import library is compatible. This field has the same format as the first 4 bytes of a resource of type `'vers'`.
- The application stack size. For an application, this field specifies the minimum size, in bytes, of the application stack. You can use the Rez constant `kDefaultStackSize` (0) to indicate that the stack should be given the default size for the current software and hardware configuration. If you determine at run time that your application needs

a larger or smaller stack, you can use the standard stack-adjusting techniques that call `GetApplLimit` and `SetApplLimit`.

- The application's library directory. For an application, this field specifies the resource ID of an alias resource (a resource of type `'alis'`) in the application's resource fork that describes the application's load directory. See "Import Library Searching" on page 3-5 for more information about load directories. For information about alias resources, see the chapter "Alias Manager" in *Inside Macintosh: Files*.
- A usage field. This field specifies the type of fragment that this record describes. The value `kIsLib` (0) indicates that the fragment is an import library. The value `kIsApp` (1) indicates that the fragment is an application. The value `kIsDropIn` (2) indicates that the fragment is an extension. The Code Fragment Manager recognizes only the values `kIsLib` and `kIsApp`. The value `kIsDropIn` is provided to allow you to put private application extensions in a file and not have the Code Fragment Manager recognize them as shared libraries.
- A location field. This field specifies the location of the fragment's container. The value `kInMem` (0) indicates that the container is in memory (usually in ROM). This value is intended for use by the Operating System; in general, you should not use it. The value `kOnDiskFlat` (1) indicates that the container is in the data fork of some file. The value `kOnDiskSegmented` (2) indicates that the container is in a resource in the resource fork of some file.
- The offset to the beginning of the fragment. The interpretation of this field depends on the value specified in the location field immediately preceding this field. If the location field has the value `kInMem`, this field is the address in memory of the beginning of the fragment. If the location field has the value `kOnDiskFlat`, this field is the number of bytes from the beginning of the data fork to the beginning of the fragment itself. You can use the Rez constant `kZeroOffset` (0) to specify an offset of 0 bytes. If the location field has the value `kOnDiskSegmented`, this field is the resource type (of type `OSType`) of the resource that contains the fragment.
- The length of the fragment. The interpretation of this field depends on the value specified in the location field immediately preceding the offset field. If the location field has the value `kInMem`, this field is the address in memory of the end of the fragment. If the location field has the value `kOnDiskFlat`, this field is the length, in bytes, of the fragment. You can use the Rez constant `kWholeFork` (0) to indicate that the fragment occupies the entire fork. If the location field has the value `kOnDiskSegmented`, this field is the sign-extended resource ID of the resource that contains the fragment.
- Reserved. The next two long integers are reserved and should be set to 0.
- The total length of the code fragment information record. This field specifies the length, in bytes, of this code fragment information record, including the fragment name and any pad bytes added to the name field.
- The fragment's name. This field is a Pascal string that indicates the name of the application or import library. This is the default name used by the debugger for this fragment. This field is padded with null bytes, if necessary, so that the information record extends to a 4-byte boundary.

# Summary of the Code Fragment Manager

## C Summary

### Constants

```
/*Gestalt selector and response bits*/
#define gestaltCFMAttr     'cfrg'   /*Code Fragment Manager attributes*/
enum {
   gestaltCFMPresent = 0            /*set if Code Fragment Mgr is present*/
};

#define kPowerPCArch       'pwpc'   /*PowerPC instruction set architecture*/
#define kMotorola68KArch   'm68k'   /*680x0 instruction set architecture*/

#define kNoLibName                  ((unsigned char *) 0)
#define kNoConnectionID             ((ConnectionID) 0)
#define kUnresolvedSymbolAddress    ((Ptr) 0x0)

enum {
   kLoadLib          = 1,           /*load fragment*/
   kFindLib          = 2,           /*find fragment*/
   kLoadNewCopy      = 5            /*load fragment with new copy of data*/
};

enum {
   kCodeSymbol       = 0,           /*a code symbol*/
   kDataSymbol       = 1,           /*a data symbol*/
   kTVectSymbol      = 2            /*a transition vector symbol*/
};

enum {
   /*selectors for fragment location record*/
   kInMem,                          /*container in memory*/
   kOnDiskFlat,                     /*container in a data fork*/
   kOnDiskSegmented                 /*container in a resource*/
};
```

## Data Types

```
typedef long                    ConnectionID;  /*connection ID number*/

typedef unsigned long           LoadFlags;     /*a flag long word*/

typedef unsigned char           SymClass;      /*symbol class*/
```

### Fragment Initialization Block

```
struct InitBlock {
   long                     contextID;     /*context ID*/
   long                     closureID;     /*closure ID*/
   long                     connectionID;  /*connection ID*/
   FragmentLocator          fragLocator;   /*fragment location*/
   Ptr                      libName;       /*pointer to fragment name*/
   long                     reserved4a;    /*reserved*/
   long                     reserved4b;    /*reserved*/
   long                     reserved4c;    /*reserved*/
   long                     reserved4d;    /*reserved*/
};
typedef struct InitBlock InitBlock, *InitBlockPtr;
```

### Fragment Location Record

```
struct FragmentLocator {
   long                     where;         /*location selector*/
   union {
      MemFragment           inMem;         /*memory location record*/
      DiskFragment          onDisk;        /*disk location record*/
      SegmentedFragment     inSegs;        /*segment location record*/
   } u;
};
typedef struct FragmentLocator FragmentLocator, *FragmentLocatorPtr;
```

### Memory Location Record

```
struct MemFragment {
   Ptr                      address;       /*pointer to start of fragment*/
   long                     length;        /*length of fragment*/
   Boolean                  inPlace;       /*is data section in place?*/
};
typedef struct MemFragment MemFragment;
```

### Disk Location Record

```
struct DiskFragment {
   FSSpecPtr               fileSpec;      /*pointer to FSSpec*/
   long                    offset;        /*offset to start of fragment*/
   long                    length;        /*length of fragment*/
};
typedef struct DiskFragment DiskFragment;
```

### Segment Location Record

```
struct SegmentedFragment {
   FSSpecPtr               fileSpec;      /*pointer to FSSpec*/
   OSType                  rsrcType;      /*resource type*/
   short                   rsrcID;        /*resource ID*/
};
typedef struct SegmentedFragment SegmentedFragment;
```

## Code Fragment Manager Routines

### Loading Fragments

```
OSErr GetDiskFragment       (FSSpecPtr fileSpec, long offset, long length,
                             Str63 fragName, LoadFlags findFlags,
                             ConnectionID *connID, Ptr *mainAddr,
                             Str255 errName);
OSErr GetMemFragment        (Ptr memAddr, long length, Str63 fragName,
                             LoadFlags findFlags, ConnectionID *connID,
                             Ptr *mainAddr, Str255 errName);
OSErr GetSharedLibrary      (Str63 libName, OSType archType,
                             LoadFlags findFlags, ConnectionID *connID,
                             Ptr *mainAddr, Str255 errName);
```

### Unloading Fragments

```
OSErr CloseConnection       (ConnectionID *connID);
```

### Finding Symbols

```
OSErr FindSymbol            (ConnectionID connID, Str255 symName,
                             Ptr *symAddr, SymClass *symClass);
OSErr CountSymbols          (ConnectionID connID, long *symCount);
OSErr GetIndSymbol          (ConnectionID connID, long symIndex,
                             Str255 symName, Ptr *symAddr,
                             SymClass *symClass);
```

## Fragment-Defined Routines

### Initializing Fragments

```
typedef OSErr ConnectionInitializationRoutine
                            (InitBlockPtr initBlkPtr);
```

### Terminating Fragments

```
typedef void ConnectionTerminationRoutine
                            (void);
```

## Result Codes

| | | |
|---|---|---|
| `fragNoErr` | 0 | No error |
| `paramErr` | –50 | Parameter error |
| `fragContextNotFound` | –2800 | Context ID is not valid |
| `fragConnectionIDNotFound` | –2801 | Connection ID is not valid |
| `fragSymbolNotFound` | –2802 | Symbol was not found in connection |
| `fragSectionNotFound` | –2803 | Section was not found |
| `fragLibNotFound` | –2804 | Library name not found in fragment registry |
| `fragDupRegLibName` | –2805 | Registered name already in use |
| `fragFormatUnknown` | –2806 | Fragment container format unknown |
| `fragHadUnresolveds` | –2807 | Loaded fragment has unacceptable unresolved symbols |
| `fragNoMem` | –2809 | Not enough memory for internal bookkeeping |
| `fragNoAddrSpace` | –2810 | Not enough memory in user's address space for section |
| `fragNoContextIDs` | –2811 | No more context IDs available |
| `fragObjectInitSeqErr` | –2812 | Order error during user initialization function |
| `fragImportTooOld` | –2813 | Import library is too old |
| `fragImportTooNew` | –2814 | Import library is too new |
| `fragInitLoop` | –2815 | Circularity in required initialization order |
| `fragInitRtnUsageErr` | –2816 | Boot library has initialization routine |
| `fragLibConnErr` | –2817 | Error connecting to library |
| `fragMgrInitErr` | –2818 | Error during Code Fragment Manager initialization |
| `fragConstErr` | –2819 | Internal inconsistency discovered |
| `fragCorruptErr` | –2820 | Fragment container is corrupted |
| `fragUserInitProcErr` | –2821 | Initialization procedure did not return `noErr` |
| `fragAppNotFound` | –2822 | No application found in `'cfrg'` resource |
| `fragArchErr` | –2823 | Fragment targeted for unacceptable architecture |
| `fragInvalidFragmentUsage` | –2824 | Fragment is used invalidly |