Mac OS Runtime Architectures

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

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, ADC will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to ADC.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures, Tables, and Listings

Chapter 1 CFM-Based Runtime Architecture 1-1

Chapter 2 Indirect Addressing in the CFM-Based Architecture 2-1

Chapter 3 Programming for the CFM-Based Runtime Architecture 3-1

Chapter 4 PowerPC Runtime Conventions 4-1

About This Book

This book describes the Mac OS runtime architecture based upon the Code Fragment Manager (CFM) as well as the original classic 68K runtime architecture.

- The CFM-based runtime architecture was originally conceived and designed to run on PowerPC-based computers running the Mac OS. A 68K implementation, called CFM-68K, was later created to allow 68K-based machines to run CFM-based code.
- The classic 68K runtime architecture is the architecture created for the original 68K-based Macintosh computer.

A **runtime architecture** is a fundamental set of rules that defines how software operates. These rules define

- how to address code and data
- how to handle and keep track of programs in memory (multiple applications and so on)
- how compilers should generate code (for example, does it allow self-modifying code?)
- how to invoke certain system services

Architectures are platform-independent, although the implementation of an architecture may vary from machine to machine depending on the features (or constraints) available.

In addition to describing the runtime architectures, this book also covers information such as calling conventions for each architecture implementation, data and register types and sizes, and details of structures encountered when building Macintosh programs (segments, fragments, and so on).

This book assumes that you are familiar with Macintosh computers and writing programs (using compilers, linkers, and so on).

What's in This Book

Information in this book is grouped by architecture type: the CFM-based or the original classic 68K runtime architecture.

The first nine chapters describe the architecture based on the Code Fragment Manager (CFM) and give details of the PowerPC and CFM-68K implementations.

- Chapter 1, "CFM-Based Runtime Architecture," gives an overview of the architecture based on the Code Fragment Manager. It includes general information about fragments, shared libraries, and the code fragment resource.
- Chapter 2, "Indirect Addressing in the CFM-Based Architecture," describes the indirect addressing model used in the CFM-based runtime architecture. It describes indirect addressing of data, the use of transition vectors, and also information about the PowerPC and CFM-68K implementations of this model.
- Chapter 3, "Programming for the CFM-Based Runtime Architecture," gives practical programming tips for writing CFM-based architecture code. It also includes information about common pitfalls that can occur.
- Chapter 4, "PowerPC Runtime Conventions," lists low-level conventions such as register volatility and usage; default data types, sizes, and alignments; and calling conventions.
- Chapter 5, "CFM-68K Runtime Conventions," lists low-level conventions such as register volatility and usage; default data types, sizes, and alignments; and calling conventions. Note that much of this information is different from the classic 68K runtime conventions.
- Chapter 6, "The Mixed Mode Manager," describes the workings of the Mixed Mode Manager, which allows transparent switching between CFM-based and classic 68K code.
- Chapter 7, "Fat Binary Programs,"describes fat programs, which can contain code for multiple runtime architectures.
- Chapter 8, "PEF Structure," gives low-level information about the structure of PEF (Preferred Executable Format) files, which are used to store both PowerPC and CFM-68K fragments. This chapter is primarily a reference for

those writing programs that generate or analyze executable code (linkers, debuggers, and so on).

- Chapter 9, "CFM-68K Application and Shared Library Structure," describes the file structure of CFM-68K applications and shared libraries in more detail. This material is primarily for those writing development tools.

Chapters 10 and 11 describe the classic 68K runtime architecture, which is the original runtime architecture for Macintosh computers.

- Chapter 10, "Classic 68K Runtime Architecture," gives a summary of the classic 68K runtime architecture, including discussions about the A5 world, segmentation, and the jump table. It also includes information on how to implement the far model (32-bit addressed) version of the classic 68K runtime architecture in MPW.
- Chapter 11, "Classic 68K Runtime Conventions," lists low-level conventions such as register volatility and usage; default data types, sizes, and alignments; and Pascal and C calling conventions.

Appendix A contains tables of terminology, constants, and data types that have been changed from earlier documentation.

Appendix B contains information about the MPW RTLib libraries, which you must use if you need to patch the Segment Manager routines in either the classic 68K far model or CFM-68K runtime environments.

A glossary and index are provided at the end of the book.

How to Use This Book

Most of the information in this book is useful for anyone programming Mac OS–based computers, but certain sections may be applicable only for programmers writing development tools or other low-level applications. In such cases, the opening paragraphs indicate the specialized nature of the information they contain.

If you are writing programs for the CFM-based runtime architecture, you should read chapters 1, 2, and 3 first in order, then move on to more specialized chapters as necessary.

If you are writing programs for the classic 68K runtime architecture, read Chapter 10, "Classic 68K Runtime Architecture," and then refer to Chapter 11, "Classic 68K Runtime Conventions," as necessary.

Related Documentation

For information on CFM-based architectures and the PowerPC implementation that is not included here, see *Inside Macintosh: PowerPC System Software*. For additional information on the classic 68K runtime architecture and implementation, consult other books in the *Inside Macintosh* series, especially the *Processes* and *Memory* volumes.

This book does not describe how to build programs. For that information you should read *Building and Managing Programs in MPW*. If you are not familiar with using MPW, read *Introduction to MPW* first.

Conventions Used in This Book

This book uses special conventions to present certain types of information. Words that require special treatment appear in specific fonts or font styles.

Special Fonts

This book uses several typographical conventions.

All code listings, reserved words, command options, resource types, and the names of actual libraries are shown in Letter Gothic `(this is Letter Gothic)`.

Words that appear in **boldface** are key terms or concepts and are defined in the glossary.

Command Syntax

This book uses the following syntax conventions:

`literal`	Letter Gothic text indicates a word that must appear exactly as shown.
italics	Italics indicate a parameter that you must replace with anything that matches the parameter's definition.
[ ]	Brackets indicate that the enclosed item is optional.
. . .	Ellipses (. . .) indicate that the preceding item can be repeated one or more times.
|	A vertical bar (|) indicates an either/or choice.

Types of Notes

This book uses three types of notes.

Note
A note like this contains information that is useful but that you do not have to read to understand the main text. ◆

IMPORTANT
A note like this contains information that is crucial to understanding the main text. ▲

▲ **WARNING**
Warnings like this indicate potential problems that you should keep in mind as you build your programs. Failure to heed these warnings could result in system crashes or other runtime errors. ▲

For More Information

The *Apple Developer Catalog* (ADC) is Apple Computer's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple computer platforms. Customers receive the *Apple Developer Catalog* featuring all

current versions of Apple development tools and the most popular third-party development tools. ADC offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *Apple Developer Catalog*, contact

Apple Developer Catalog
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
World Wide Web	http://www.devcatalog.apple.com

CFM-Based Runtime Architecture

Contents

The CFM-based runtime architecture relies on fragments and the Code Fragment Manager (CFM) for its operation. This architecture has been implemented as the default architecture for PowerPC-based Mac OS computers and an optional one, CFM-68K, for 68K-based machines. The key concepts are identical for both implementations, so you should read this chapter if you plan to write either PowerPC or CFM-68K code.

In addition, you should read Chapter 2, "Indirect Addressing in the CFM-Based Architecture," which contains more information related to the CFM-based architecture. Chapter 3, "Programming for the CFM-Based Runtime Architecture," contains additional practical information that you may find useful when writing CFM-based programs.

For specific information about the implementation of the CFM-based architecture on the PowerPC and 68K microprocessors, you should read the following chapters:

- Chapter 4, "PowerPC Runtime Conventions," for PowerPC information
- Chapter 9, "CFM-68K Application and Shared Library Structure," and Chapter 5, "CFM-68K Runtime Conventions," for CFM-68K information

Overview

In the CFM-based architecture, a **fragment** is the basic unit of executable code and its associated data. All fragments share fundamental properties such as basic structure and method of addressing code and data. The major advantage of a fragment-based architecture is that a fragment can easily access code or data contained in another fragment. For example, a fragment can import routines or data items from another fragment or export them for another fragment's use. In addition, fragments that export items may be shared among multiple clients.

Note
The term *fragment* is not intended to suggest that the block of code and data is in any way either small, detached, or incomplete. Fragments can be of virtually any size, and they are complete, executable entities. The term *fragment* was chosen to avoid confusion with the terms already used in *Inside Macintosh* volumes to describe executable code (such as *component* and *module*). ◆

The **Code Fragment Manager** handles fragment **preparation,** which involves bringing a fragment into memory and making it ready for execution. Fragments can be grouped by use into applications and shared libraries, but fundamentally the Code Fragment Manager treats them alike.

Fragment-based **applications** are launched from the Finder. Typically they have a user interface and use event-driven programming to control their execution.

A **shared library,** however, is a fragment that exports code and data for use by other fragments. Unlike a traditional static library, which the linker includes in the application during the build process, a shared library remains a separate entity. For a shared library, the linker inserts a reference to an imported function or data item into the client fragment. When the fragment is prepared, the Code Fragment Manager creates incarnations of the shared libraries required by the fragment and binds all references to imported code and data to addresses in the appropriate libraries. A shared library is stored independently of the fragment that uses it and can therefore be shared among multiple clients.

Note
Shared libraries are sometimes referred to as *dynamically linked libraries* (DLLs), since the application and the externally referenced code or data are linked together dynamically when the application launches. ◆

Using a shared library offers many benefits based on the fact that its code is not directly linked into one or more fragments but exists as a separate entity that multiple fragments can address at runtime. If you are developing several CFM-based applications that have parts of their source code in common, you should consider packaging all the common code into a shared library.

Here are some ways to take advantage of shared libraries:

- An application framework can be packaged as a shared library. This potentially saves a good deal of disk space because that library resides only

once on disk—where it can be addressed by multiple applications—rather than being linked physically into numerous applications.

- System functions and tools, such as OpenDoc, can be packaged as shared libraries.
- Updates and bug fixes for a single library can be released without the need to recompile and send copies of all the applications that use the library.

Shared libraries come in two basic forms:

- **Import libraries.** These contain code and data that your application requires to run. The Code Fragment Manager automatically prepares these libraries at runtime. Import libraries do not occupy application memory but are stored separately.
- **Plug-ins.** These are libraries that provide optional services, such as a spelling checker for a word processor. The application must make explicit calls to the Code Fragment Manager to prepare these libraries and must then find the symbols associated with the libraries. Plug-ins are sometimes referred to as *drop-in additions* or *extensions.*

Note
Although the terms are similar, *shared library* and *import library* are not interchangeable. An import library is a shared library, but a shared library is not necessarily an import library. ◆

In the CFM-based runtime architecture, the Code Fragment Manager handles the manipulation of fragments. Some of its functions include

- mapping fragments into memory and releasing them when no longer needed
- resolving references to symbols imported from other fragments
- providing support for special initialization and termination routines

Fragments can be shared within a process or between two or more processes. A **process** defines the scope of an independently-running program. Typically each process contains a separate application and any related plug-ins.

The physical incarnation of a fragment within a process is called a **connection.** A fragment may have several unique connections, each local to a particular process. Each connection is assigned a **connection ID.** For more information on how the Code Fragment Manager groups connections into functional programs, see “Closures” (page 1-6).

Fragments are physically stored in **containers,** which can be any kind of storage area accessible by the Code Fragment Manager. For example, in System 7 the system software import library `InterfaceLib` is stored in the ROM of a PowerPC-based Macintosh computer. Other import libraries are typically stored in files of type `'shlb'`. Fragments containing executable code are usually stored in the data fork of a file, although it is possible to store a fragment as a resource in the resource fork. For more information about container storage, see "Fragment Storage" (page 1-24), and Chapter 8, "PEF Structure."

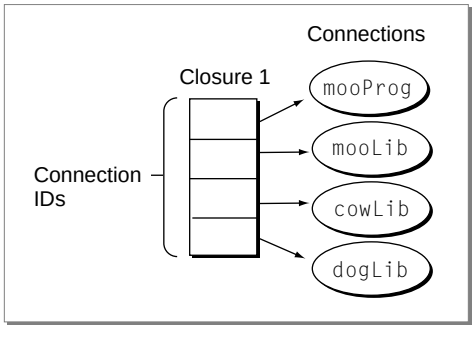
Closures

The Code Fragment Manager uses the concept of a **closure** when handling fragments. A closure is essentially a set of connection IDs that are grouped according to the order they are prepared. The connections represented by a closure are the **root fragment,** which is the initial fragment the Code Fragment Manager is called to prepare, and any import libraries the root fragment requires to resolve its symbol references.

During the fragment preparation process, the Code Fragment Manager automatically prepares all the connections required to make up a closure. This process occurs whether the Code Fragment Manager is called by the system (application launch) or programmatically from your code (for example, when preparing a plug-in).

Figure 1-1 shows a set of connections that make up a closure.

Figure 1-1 A closure

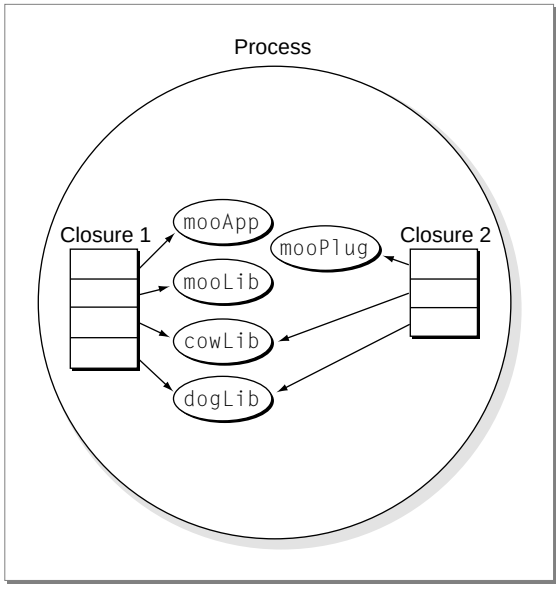


Each closure is assigned a **closure ID.**

A fragment may be used in more than one process, but a separate connection is created for each process. For example, if two applications require the standard C library, a separate connection is created for each one.

Connections may be shared among closures within a process if your application calls the Code Fragment Manager to prepare a plug-in. Figure 1-2 shows the closure of an application and a closure of a plug-in sharing a fragment within a process.

Figure 1-2 Multiple closures in a process

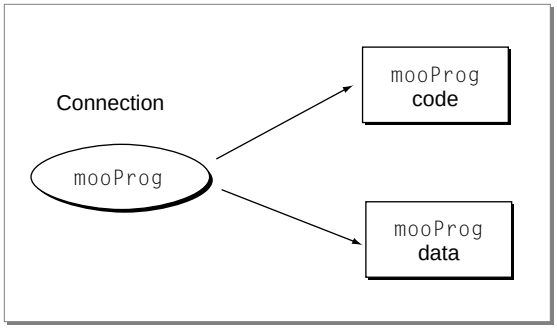


The Code Fragment Manager does not create new connections when sharing a fragment within a process but uses the ones that are currently available. Therefore it is possible for separate closures in the same process to refer to a fragment by the same connection ID. Connections are not shared across processes, however, so new connections (and connection IDs) are created when a fragment appears in another process.

Code and Data Sections

Each connection has **sections** associated with it that contain either code or data as shown in Figure 1-3.

Figure 1-3 Sections associated with a connection

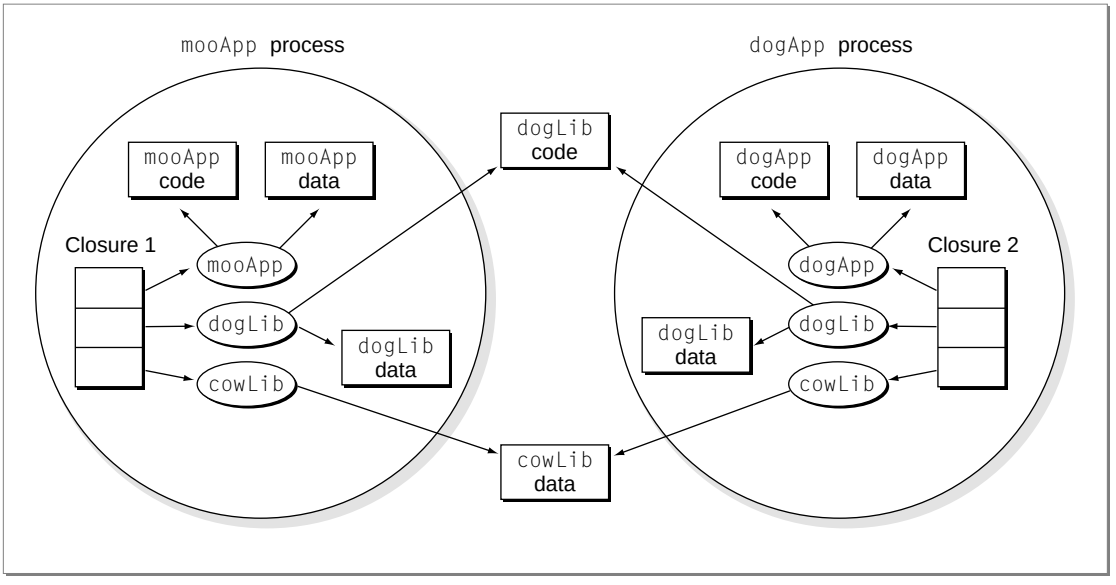


Within a process, a connection is generally shared between multiple closures, and therefore both code and data sections are shared. Fragments that are used in multiple processes share their code, but they have the following choices for sharing their data:

- Systemwide (or global) instantiation. The Code Fragment Manager allocates a single copy of the library's global data, and all connections for a particular fragment share that data.
- Per-process instantiation. The Code Fragment Manager allocates one copy of the library's global data for each process. Each connection can access only its own copy of the data.

In Figure 1-4, fragment `cowLib` is globally shared while fragment `dogLib` is shared per-process.

Figure 1-4 Fragments shared between processes



In most cases, per-process sharing is preferred over systemwide sharing. For more information about systemwide sharing, see "Systemwide Sharing and Data-Only Fragments," beginning on page 3-24.

Each library determines how its global data is to be shared, and this information is stored in the library at link time. The library developer can indicate either systemwide or per-process data instantiation for each separate data section in a library.

Reference Counts

The Code Fragment Manager keeps a **reference count** for every connection currently in a process. This value indicates the number of closures that reference the connection. For example, in Figure 1-2, the shared fragments `dogLib` and `cowLib` each have a reference count of 2. If the plug-in is released, the reference count for each would be decremented. When the reference count of a connection drops to zero, the connection is not part of any closure and the Code Fragment Manager is free to release it if necessary.

The Code Fragment Manager also keeps similar count values for each section of a shared fragment that indicates the number of connections associated with it. For example, in Figure 1-4, the code section for `dogLib` has a reference count of 2, and the two data sections for `dogLib` each have a reference count of 1. If the process containing `dogApp` terminates, the reference count for the `dogLib` data section in that process drops to zero, so the Code Fragment Manager can release the section. The reference count for the code section only drops to 1, however, so it remains in memory.

Finally, the Code Fragment Manager also keeps track of the number of connections associated with a given fragment container. If a fragment container has no connections associated with it, the Code Fragment Manager can release the container from memory.

Using Code Fragment Manager Options

If you prepare and release fragments explicitly from your code, you should be aware of the different options available. These options are available (as the `CFragLoadOptions` parameter) for the following Code Fragment Manager routines:

- `GetSharedLibrary`
- `GetDiskFragment`
- `GetMemFragment`

If you are calling one of these Code Fragment Manager routines to prepare a plug-in, you should generally specify the `kReferenceCFrag` option when invoking it. The Code Fragment Manager then prepares the fragment (and any required import libraries) if a connection is not already present, adds a new closure, and increments the reference count of any import libraries that the new closure shares with others already in memory. The Code Fragment Manager then returns a closure ID which you can use to access the symbols in the closure.

IMPORTANT

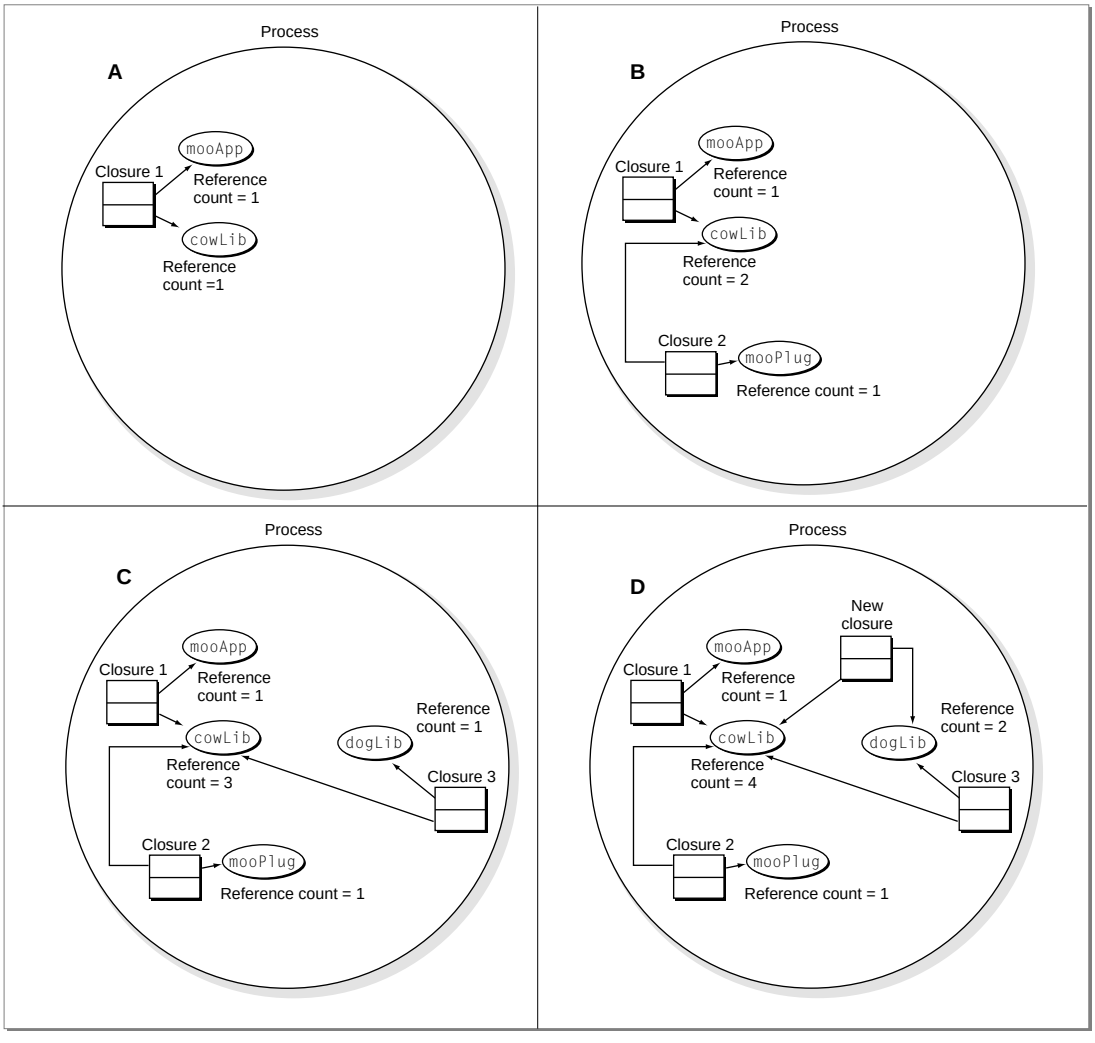
Code Fragment Manager routines that do not create a new closure return a connection ID rather than a closure ID. The Code Fragment Manager increments reference counts on existing connections only when a new closure is created. ▲

You can also use the `kReferenceCFrag` option to gain access to symbols in an already instantiated connection. For example, say you have an application `mooApp` as shown in Figure 1-5 A. The application `mooApp` prepares the plug-in `mooPlug` as shown in B, and `mooPlug` sometime later programmatically prepares the shared library `dogLib` (shown in C). If you wanted to access the symbols in `dogLib` from `mooApp`, you could do so by calling the Code Fragment Manager to prepare `dogLib` using the `kReferenceCFrag` option. The Code Fragment Manager adds a new closure and increases reference counts to reflect the presence of the new closure. Figure 1-5 D shows the effect of using `kReferenceCFrag` to prepare the shared library `dogLib`, which requires the library `cowLib`.

Note

`kReferenceCFrag` was previously called `kLoadCFrag`. ◆

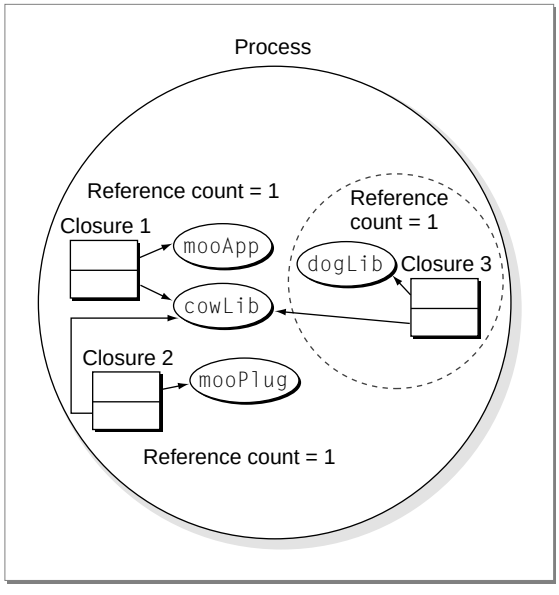
Figure 1-5 Using `kReferenceCFrag`



In some cases you may only want to determine if a connection associated with a fragment exists. In such cases, you can use the `kFindCFrag` option to return a connection ID of an existing connection. However, using the `kFindCFrag` option does not add a closure or increase the connection's reference count. You can

theoretically access the symbols it contains, but if the reference count drops to 0, the Code Fragment Manager might release the connection while your program is still using it. For example, in Figure 1-6, say that `mooApp` prepares the plug-in `mooPlug`, and `mooPlug` programmatically prepares the shared library `dogLib`. Later, `mooApp` uses `kFindCFrag` to access symbols in `dogLib`. If `mooPlug` releases `dogLib`, then any references to symbols in `dogLib` are left dangling.

Figure 1-6 Using `kFindCFrag`

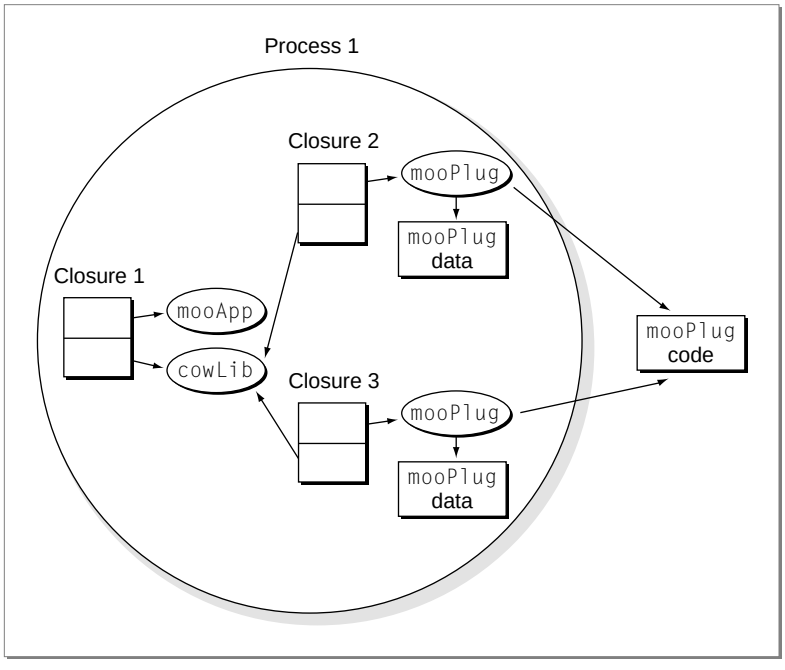


Another useful option is `kPrivateCFragCopy`. Using the `kPrivateCFragCopy` option when calling a Code Fragment Manager routine, you can create a new connection for each request to prepare the fragment, even if the same application makes multiple preparation requests. That is, you can have multiple connections (each with its own private data section) from the same shared library that all serve the same client fragment. Such a connection is called a **private connection.** A fragment prepared in this manner, however, is not visible as an import library (that is, the Code Fragment Manager does not recognize its name as an import library and you cannot find it using the `GetSharedLibrary` routine or the `kFindCFrag` option.)

Note
A private connection is also known as a *per-load instantiation.* ◆

For example, the application `mooApp` in Figure 1-7 has created two "copies" of the plug-in `mooPlug`. Each instance of `mooPlug` has its own data, but they all share the same code. Note that `cowLib` is not duplicated for each instance of `mooPlug`; any import libraries that are part of a private connection's closure are treated normally.

Figure 1-7 Using private connections



You can specify a private connection, for example, if you have a communications application that uses a shared library to implement a tool for connecting to a serial port. By requesting private connections, you can ensure that your tool can connect to two or more serial ports simultaneously by maintaining separate copies of the tool's data. The tool itself can then be ignorant of how many ports it is handling.

Preparing a Closure

When the Code Fragment Manager is called to prepare a fragment, it prepares the closure associated with the fragment to ensure that the fragment can access all its imported symbols during execution. This preparation process involves the following steps:

1. Determine the closure associated with the root fragment (an application or a plug-in, for example). The Code Fragment Manager does the following to determine the closure:
   - Finds compatible versions of all the import libraries the root fragment requires. Note that some import libraries may themselves depend on other import libraries.
   - Brings into memory any fragments that do not currently have connections to them.
   - Assigns connection IDs to any new connections and assigns a closure ID to the new closure.

   See "Searching for Import Libraries," beginning on page 1-16, for information about the library search procedure and "Checking for Compatible Import Libraries," beginning on page 1-19, for information about how the Code Fragment Manager checks version compatibility.
2. Instantiate (or locate, if already present) code and data sections for each connection in the closure. This procedure assigns actual addresses to the sections and, consequently, assigns addresses to all exported symbols.
3. Resolve all imported symbols. For every imported symbol required by the new connections in the closure, the Code Fragment Manager finds the corresponding exported symbol address and stores it in an internal lookup table.
4. Do relocations. Using the lookup table compiled in step 3 and the section addresses determined in step 2, the Code Fragment Manager replaces all references to imported symbols (and any other pointer-based symbol references) in the closure with actual addresses. See Chapter 2, "Indirect Addressing in the CFM-Based Architecture," and the section "Relocations," beginning on page 8-21, for more details.
5. Execute initialization functions (if any exist).

6. Return the closure ID and main symbol to the caller.

IMPORTANT

These steps apply for any fragment the Code Fragment Manager is called to prepare (including plug-ins). ▲

In general, if the Code Fragment Manager cannot complete any step, then the preparation fails and an error is returned. The only special case is when certain libraries or symbols have been declared weak. A weak library or symbol is one that is marked as being optional; the preparation process can continue even if the library or symbol is not available. However, once a weak library or symbol is determined to be present, it is handled normally for the rest of the preparation process. For example, if a weak library is available but cannot be prepared properly for some reason, the whole closure preparation fails. See "Weak Libraries and Symbols," beginning on page 3-11, for more information.

Note

A weak library is determined to be present or not present in step 1 of the preparation process. Weak symbols are determined in step 3. ◆

Searching for Import Libraries

When the Code Fragment Manager is called to prepare a fragment, if the fragment requires other import libraries to complete the closure, the Code Fragment Manager goes through an ordered search process to find physical copies of those libraries. For example, the Code Fragment Manager can search folders containing the application or the root fragment as well as a common folder specially designated to hold shared libraries.

Currently the Code Fragment Manager looks for files that contain a resource of type `'cfrg'`. The `'cfrg'0` resource identifies the fragment name of the import library. There can be more than one fragment name listed in a single `'cfrg'0` resource. This might happen if there are multiple import libraries contained in a single file or if a single import library or application is to be identified by more than one name. Fragments are typically stored in the data fork, although it is possible to store a fragment in a resource. In either case, the `'cfrg'0` resource points to the location of the fragment within the file. For more information about the `'cfrg'0` resource, see "The Code Fragment Resource," beginning on page 1-25.

Once the Code Fragment Manager finds a library that is compatible with the fragment it's preparing, it stops searching and resolves imports in the fragment to code or data in that library. If it reaches the end of its search without finding a compatible library, the fragment preparation fails.

Note
Because the Code Fragment Manager is searching for the import library by name, the file containing the library must have a `'cfrg'0` resource. However, you can prepare fragments that do not contain a `'cfrg'0` resource by calling Code Fragment Manager routines from your program. See "Calling the Code Fragment Manager," beginning on page 3-3, for more information. ◆

In System 7 through 7.5, the search process for import libraries is as follows:

1. **Check connections in the same process to see if a connection for the import library already exists.**

   If the connection is already in use in another closure, then the Code Fragment Manager can simply increment its reference count and use it.

   If the existing connection is associated with an incompatible version of the import library, the preparation fails. In all the steps that follow, however, finding an incompatible import library version merely causes the Code Fragment Manager to move to the next step in the search procedure. See "Checking for Compatible Import Libraries" (page 1-19) for more information about how the Code Fragment Manager checks for compatible libraries.

2. **Check the folder containing the root fragment.**

   If the root fragment folder is not the same as the application folder, the Code Fragment Manager searches here first. The Code Fragment Manager looks only in the top level of the folder, not in any subfolders contained within it.

3. **Check the file containing the application.**

   Since a file can contain multiple fragments, the file containing the application fragment may also contain import library fragments.

4. **Check the application subfolder.**

   When you build your application, you can designate a library folder for the Code Fragment Manager to search for import libraries. For more information, see "The Code Fragment Resource," beginning on page 1-25.

5. **Check the folder containing the application.**

   The Code Fragment Manager looks only in the top level of the application folder, not in any subfolders contained within it.

6. **Check the Extensions folder.**

   The Extensions folder usually contains import libraries used by multiple applications (libraries for QuickTime, for example). The Code Fragment Manager searches the Extensions folder and one level of folders inside the Extensions folder.

7. **Check the ROM registry.**

   The ROM registry keeps track of all import libraries that are stored in the ROM of a Mac OS–based computer. The Mac OS registers all ROM-based import libraries in this registry at system startup time.

8. **Check the file registry.**

   The final stage of the search path is a file and directory registry that the Code Fragment Manager maintains internally. This registry, which is currently reserved for system use, 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.

In System 7.6, the Code Fragment Manager combines steps 6, 7, and 8, searching all three locations and choosing the import library that best fits the compatibility requirements.

The Code Fragment Manager searches a folder by looking for files of type `'shlb'` that contain a resource of type `'cfrg'`. Within a folder, the Code Fragment Manager also looks for alias files of type `'shlb'` and resolves them to their targets.

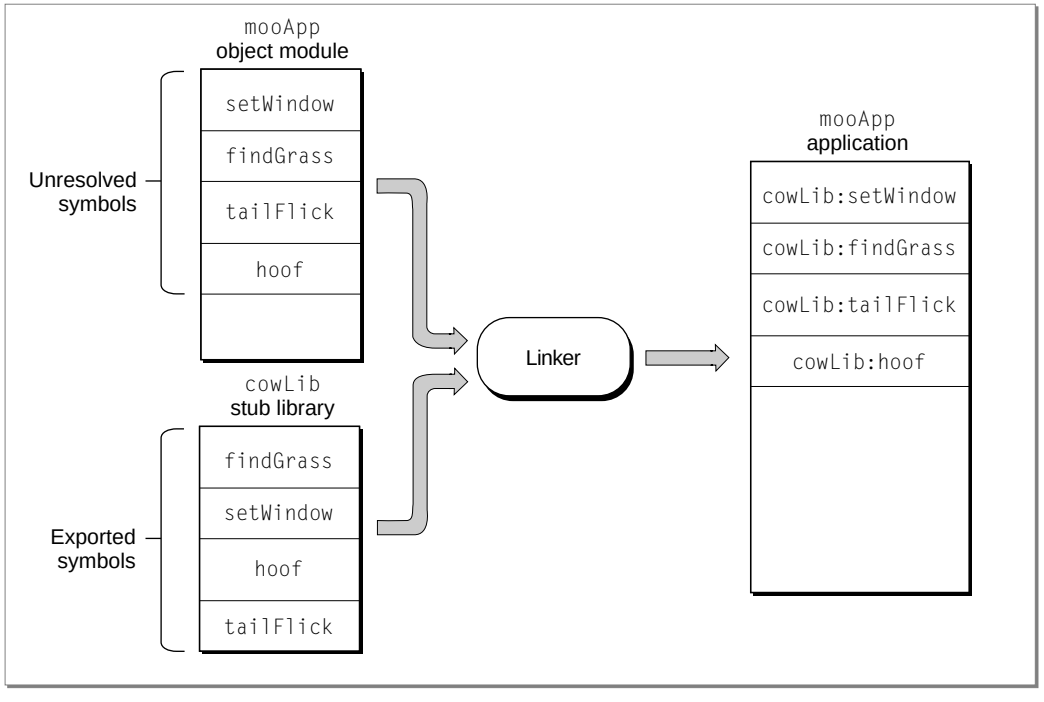
At any stage, the Code Fragment Manager selects the one import library of all those available to it that best satisfies its compatibility version checking. If an import library meets the relevant criteria, the library search stops. Otherwise, the search continues to the next stage. If the final stage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

Checking for Compatible Import Libraries

Checking compatibility between a client fragment and an import library essentially means checking for an intersection between the version range required by the client fragment and the range supported by the import library.

When building a fragment that requires an import library, you must supply information in a **definition stub library** that defines the library's API. A **stub library** contains symbol definitions but does not contain actual code. The linker uses definition stub libraries to associate imported symbols with particular import libraries. Figure 1-8 shows an application linking to a definition stub library to produce the completed application. A reference such as `cowLib:setWindow` means that the symbol `setWindow` can be found in the import library `cowLib`.

Figure 1-8 Linking to a definition stub library

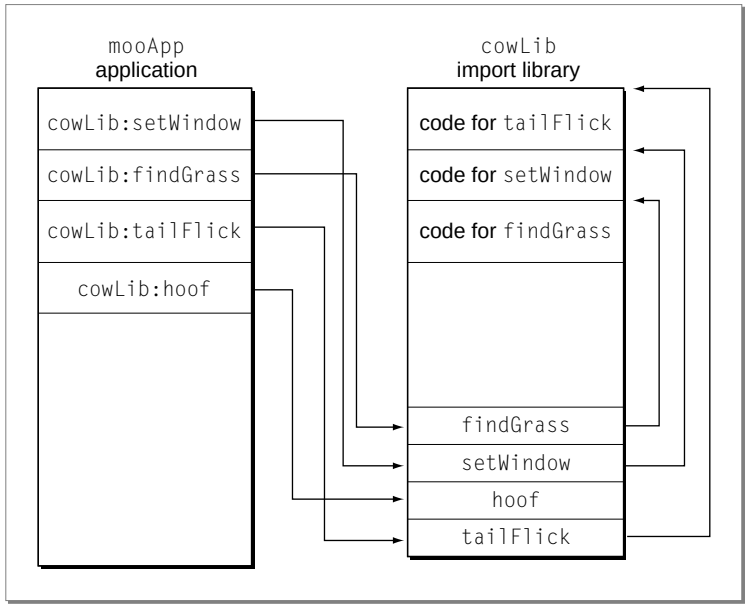


An import library that provides implementation code is dynamically linked to the client fragment by the Code Fragment Manager during the preparation process. This library (sometimes called the **implementation library**) must be fully functional.

Note
Since an implementation library contains symbol definitions, the implementation library can act as a definition stub library at link time. ◆

Figure 1-9 shows an implementation library bound to the application at runtime.

Figure 1-9 Using the implementation version of a library at runtime



The definition stub library may not be the same version as the implementation library (one may be an earlier version, for example), so the Code Fragment Manager must check to make sure that they are compatible. Generally the libraries are compatible if the library used at runtime can satisfy the programming interface defined for it during the build process.

When building an import library, you determine compatibility by defining version numbers. You should set three version numbers (usually by specifying linker options) for use in version checking:

- the current version number of the library you are creating
- the old implementation version number, which is the oldest version of this library available at runtime that supports the client's needs
- the old definition version number, which is the oldest version of the library defined for the client fragment that is supported by the library you are creating

When building a client fragment, the linker stores the current version and old implementation version numbers of the import library in the client. Later, when the Code Fragment Manager prepares the client fragment, it uses this information to check for a compatible import library.

Table 1-1 shows two different versions of an import library `cowLib` and their version numbers.

Table 1-1 Two import libraries and their version numbers

Name	Current version number	Old definition version number	Old implementation version number
`cowLib 13`	13	9	10
`cowLib 16`	16	12	14

IMPORTANT

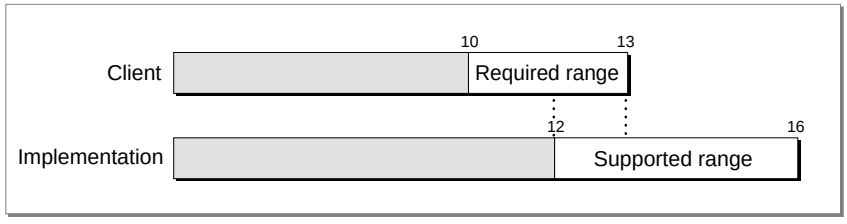
The current version number must always be greater than or equal to both the old definition version number and the old implementation version number. ▲

If you build an application `mooApp` with `cowLib 13` and attempt to run with `cowLib 16`, the compatibility ranges are as follows:

- `cowLib 13` is not compatible with implementations of `cowLib` earlier than version 10 (its old implementation number is set to 10).
- `cowLib 16` (present in the Extensions folder, for example) is not compatible with definitions of `cowLib` earlier than version 12 (its old definition version number is set to 12).

Figure 1-10 shows the compatibility ranges graphically.

Figure 1-10 Library versions compatible with each other



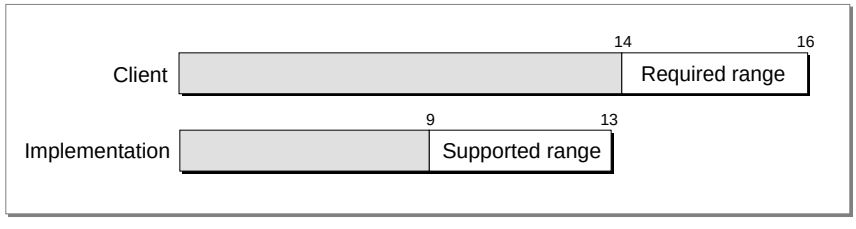
The presence of an overlap between the two areas of compatibility indicates that the two import libraries are compatible in this case.

However, reversing the roles of the two libraries (building with `cowLib 16`, executing with `cowLib 13`) results in a different set of compatibility ranges as follows:

- `cowLib 16` is not compatible with implementations of `cowLib` earlier than version 14 (the old implementation version number is 14).
- `cowLib 13` is not compatible with definitions of `cowLib` earlier than version 9 (the old definition version number is 9).

In this case the two libraries are incompatible, as shown in Figure 1-11.

Figure 1-11 Library versions incompatible with each other



The library `cowLib 16` may (for example) include additional routines that are not supported by versions older than 14.

The Code Fragment Manager uses the algorithm shown in Listing 1-1 for checking import library compatibility. It uses this algorithm to check the compatibility of the fragment being prepared with *all* the import libraries from which it imports code and data.

Listing 1-1 Pseudocode for the CFM 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);
```

The fact that only one instance of an import library can appear in a given process may cause versioning conflicts. For example say an application `mooApp` uses the import library `mooLib`. If `mooApp` loads a plug-in `mooPlug` that also requires `mooLib`, then the Code Fragment Manager uses the available connection for `mooLib`. If this version is not compatible with the version required by the plug-in, then the preparation of the plug-in fails. (This failure occurs even if you designated `mooLib` to be weak.)

The Structure of Fragments

Every fragment can contain separate code and data sections. A code or data section can be up to 4 GB in size. Code and data sections do not have to be contiguous in memory.

Note
Since all fragments can contain both code and data sections, any fragment can contain global variables. ◆

A **code section** contains position-independent executable code (that is, code that is independent of its own memory location and the location of its associated data). Code sections are read-only, so fragments can be stored in ROM or file-mapped and paged in from disk as necessary.

A **data section** is typically allocated in the application heap. Each data section may be instantiated multiple times, creating a separate copy for each connection associated with the fragment. See "Closures," beginning on page 1-6, for more details. An import library's data section may also be placed into the system heap or temporary memory (when systemwide instantiation is selected).

Although a fragment's code and data sections can be located anywhere in memory, those sections cannot be moved within memory once they are prepared. The Code Fragment Manager must resolve any dependencies a fragment might have on other fragments, and this preparation involves placing pointers to imported code and data into the fragment's data section. To avoid having to prepare fragments in this way more than once, the Mac OS requires that a prepared fragment remain stationary as long as it stays in memory.

Note
Accelerated resources, which model the behavior of classic 68K resources, do not have to be fixed in memory between calls. ◆

Fragment Storage

The physical storage unit for a fragment is called its container. A container can be any logically contiguous piece of storage, such as the data fork of a file (or some portion thereof), the Macintosh ROM, or a resource. The System 7 version of the Code Fragment Manager recognizes two container formats, the Preferred Executable Format (PEF) and the Extended Common Object File Format (XCOFF). Note that compatibility with these formats is not a requirement of the CFM-based architecture, and it may change in the future.

- The **Preferred Executable Format (PEF),** as defined by Apple Computer, is the current standard executable file format for CFM-based architectures. PEF provides full support of a fragment's attributes. See Chapter 8, "PEF Structure," for more details of this format.

- The **Extended Common Object File Format** (XCOFF) is a refinement of the Common Object File Format (COFF), the standard executable file format on many UNIX®-based computers. XCOFF containers tend to be larger than their PEF counterparts and often take longer to load into memory. XCOFF containers do not support initialization or termination routines, and they do not store version information for import library compatibility checks. XCOFF is supported on Mac OS–based computers primarily because early development tools produced executable code in the XCOFF format.

Note
Not all object code in the XCOFF format can execute on Mac OS–based computers. Any XCOFF code that uses UNIX-style memory services or that otherwise depends on UNIX features does not execute correctly on Mac OS–based computers. XCOFF output from a compiler also does not execute. ◆

The Code Fragment Resource

If the Code Fragment Manager is to search for a fragment by name, the file containing the fragment must contain a **code fragment resource** in the resource fork. A code fragment resource is a resource of type `'cfrg'` with ID 0 (`'cfrg'0` resource).

The code fragment resource has the form shown in Listing 1-2.

Listing 1-2 The code fragment resource

```
struct CFragResource {
    UInt32                  reservedA;  /* must be zero! */
    UInt32                  reservedB;  /* must be zero! */
    UInt16                  reservedC;  /* must be zero! */
    UInt16                  version;
    UInt32                  reservedD;  /* must be zero! */
    UInt32                  reservedE;  /* must be zero! */
    UInt32                  reservedF;  /* must be zero! */
    UInt32                  reservedG;  /* must be zero! */
    UInt16                  reservedH;  /* must be zero! */
    UInt16                  memberCount;
    CFragResourceMember     firstMember;
};
```

- The `version` field indicates the version of the code fragment resource. The current version is `1`.
- The `memberCount` field indicates how many fragment entries (`'cfrg'0` entries) are described by this resource.
- Each entry of type `CFragResourceMember` describes a fragment entry, listing the type of fragment, its name, location, and so on.

Since the `'cfrg'0` resource is an array, it is possible to store information for several fragments in one file. The fragments remain separate and the Code Fragment Manager can prepare them independently, but they can be shipped and marketed as a single file. In addition, the code fragment resource can point to fragments of multiple architectures, allowing you to create fat applications and shared libraries that can execute on multiple platforms. See Chapter 7, "Fat Binary Programs," for more information.

Note
Typically you can use a development tool (such as MergeFragment in MPW) to place multiple fragments in a file. ◆

The structure of the code fragment resource is identical for all fragment types, although some of the field values may differ. Field values in the code fragment resource are determined and set at link time, but some may be changed later using a resource editor (such as ResEdit). Field values are defined in `CodeFragments.h`.

A code fragment resource entry has the form shown in Listing 1-3.

Listing 1-3 A code fragment resource entry

```
struct CFragResourceMember {
    CFragArchitecture   architecture;
    UInt16              reservedA;      /* zero */
    UInt8               reservedB;      /* zero */
    UInt8               updateLevel;
    CFragVersionNumber  currentVersion;
    CFragVersionNumber  oldDefVersion;
    CFragUsage1Union    uUsage1;
    CFragUsage2Union    uUsage2;
    CFragUsage          usage;
```

```
    CFragLocatorKind    where;
    UInt32              offset;
    UInt32              length;
    UInt32              reservedC;      /* zero */
    UInt32              reservedD;      /* zero */
    UInt16              extensionCount; /* number of extensions */
    UInt16              memberSize;     /* total size in bytes */
    unsigned char       name [kDefaultCFragNameLen];
};
```

- The `architecture` field indicates the runtime environment of the fragment. Current values for this field are as follows:
  - `kPowerPCCFragArch` for the PowerPC runtime environment
  - `kMotorola68KCFragArch` for the CFM-68K runtime environment
  - `kCompiledCFragArch`, which is conditionally set at compile time. For example, if you are compiling for the PowerPC runtime environment, this value is set to `kPowerPCCFragArch`. You can specify this value in source code that is used for both PowerPC and CFM-68K builds.
- The `updateLevel` field indicates whether this fragment is a base fragment or one created to update another fragment. This field typically has the value `kIsCompleteCFrag` to indicate a base fragment.
- The next two fields, `currentVersion` and `oldDefVersion`, store the current and oldest definition version numbers that the Code Fragment Manager relies on for checking compatibility with client fragments. If a fragment does not export any symbols, it does not need to check compatibility, and these values can be ignored.
- The `uUsage1` field contains a union defined as

```
union CFragUsage1Union {
        UInt32  appStackSize;
    };
```

  If the fragment is an application, `appStackSize` indicates the application stack size. Typically `appStackSize` has the value `kDefaultStackSize`.

- The `uUsage2` field contains a union defined as

```
union CFragUsage2Union {
        SInt16  appSubdirID;
    };
```

If the fragment is an application, `appSubdirID` indicates the library directory. By default, the Code Fragment Manager searches the folder containing the application and the Extensions folder when looking for import libraries, but you can specify a library directory in addition to the default search directories (see "Searching for Import Libraries," beginning on page 1-16, for more information). If you do not specify a library directory, this field has the value `kNoAppSubFolder`. In System 7, if you want to add another library directory, you must change this field to the resource ID of an alias resource (a resource of type `'alis'`) in the application's resource fork. This resource should describe the application's library directory. For more information about alias resources, see the chapter "Alias Manager" in *Inside Macintosh: Files*.

- The `usage` field indicates the type of fragment. Possible values are as follows:
  - `kApplicationCFrag` for an application
  - `kImportLibraryCFrag` for an import library
  - `kDropInAdditionCFrag` for a plug-in
- The `where` field indicates where the fragment is located. Possible values are as follows:
  - `kDataForkCFragLocator` if the fragment is in the data fork
  - `kMemoryCFragLocator` if the fragment is stored in ROM
  - `kResourceCFragLocator` if the fragment is stored in a resource
- The next two fields, `offset` and `length`, indicate the starting and ending offsets of the fragment container. For example, the values `kZeroOffset` and `kCFragGoesToEOF` indicate that the container for the fragment starts at the beginning of the data fork and ends at the end of the data fork.

  If the fragment is stored in a resource, the `offset` field describes the type of resource, and the `length` field contains the resource ID number.
- The field `extensionCount` indicates the number of extensions. See "Extensions to Code Fragment Resource Entries" (page 1-29) for more information.
- The field `memberSize` contains the total size, in bytes, of the code fragment resource entry. This size includes any extensions.
- The `name` field contains the name of the fragment.

Extensions to Code Fragment Resource Entries

The basic code fragment resource entry structure shown in Listing 1-3 is used for most applications and shared libraries. However, a code fragment resource entry can also contain one or more extensions, which appear directly after the fragment name. Such an extended code fragment resource entry stores additional information about the fragment that may be used by third-party software. For example, while the regular entry might simply indicate that a fragment `mooLib` is an import library, an extension could also indicate that it is a SOM class library that inherits from the class `cow`.

Note
A code fragment resource can contain any combination of extended and regular entries. ◆

Padding is added after the `name` field to begin the extensions on a 4-byte boundary (the length byte of the `name` string does not include this padding). All extensions must be aligned on 4-byte boundaries, with padding added after each if necessary. The `memberSize` field includes any padding added after the last extension.

An extension to the code fragment resource has the form shown in Listing 1-4.

Listing 1-4 Structure of a sample code fragment resource extension

```
struct CFragResourceSearchExtension {
    CFragResourceExtensionHeader    extensionHeader;
    ExtensionData                   data [1];
};
```

The `extensionHeader` field contains a data structure defined as shown in Listing 1-5.

Listing 1-5 The code fragment resource extension header

```
struct CFragResourceExtensionHeader {
    UInt16  extensionKind;
    UInt16  extensionSize;
};
```

- The `extensionKind` field defines the type of extension. Each type defines the format of the information contained in the extension. Currently only one is defined (`extensionKind` = `30EE`).
- The `extensionSize` field specifies the total size, in bytes, of this extension, including any padding necessary to round the extension to a 4-byte boundary. This size added to the offset of the extension gives the offset of the next extension (if any).

The information that follows the `extensionHeader` field depends on the value of `extensionKind`. As an example, Listing 1-6 shows the format of the code fragment resource extension of type `30EE`.

Listing 1-6 A code fragment resource extension of type `30EE`

```
struct CFragResourceSearchExtension {
    CFragResourceExtensionHeader    extensionHeader;
    OSType                          libKind;
    unsigned char                   qualifiers [1];
};
```

- The `libKind` field indicates the type of fragment. Currently defined values are as follows:
  - `kFragDocumentPartHandler` for a part handler
  - `kFragSOMClassLibrary` for a SOM class library
  - `kFragInterfaceDefinition` for an interface definition library
  - `kFragComponentMgrComponent` for a component used by the Component Manager
- After the `libKind` field, you can define up to four Pascal-style strings in the `qualifiers` field. The values of these strings depend on the `libKind` field. The currently defined values are as follows:
  - For type `kFragDocumentPartHandler`, the first qualifier indicates the handler type. The second qualifier indicates the handler subtype (if any).
  - For type `kFragSOMClassLibrary`, the first qualifier indicates the base class.
  - For type `kFragInterfaceDefinition`, the first qualifier indicates the interface definition name.

- For type `kFragComponentMgrComponent`, the first qualifier indicates the component type. The second qualifier indicates the component subtype.

For any extension, the fourth qualifier can hold the name of the fragment. Unlike the string in the `name` field, this string is visible to the client fragment.

Sample Code Fragment Resource Entry Definitions

This section contains examples of the most common types of code fragment resource entries.

A PowerPC Application 'cfrg' 0 Resource Definition

Listing 1-7 shows an example of a `'cfrg'0` resource definition for a PowerPC application.

Listing 1-7 A sample `'cfrg'0` resource for a PowerPC runtime application

```
#include "CodeFragmentTypes.r"
resource 'cfrg' (0) {
    {
        kPowerPCCFragArch,      /* runtime environment */
        kIsCompleteCFrag,       /* base-level library */
        kNoVersionNum,          /* current version number*/
        kNoVersionNum,          /* oldest definition version number */
        kDefaultStackSize,      /* use default stack size */
        kNoAppSubFolder,        /* no library directory */
        kApplicationCFrag,      /* fragment is an application */
        kDataForkCFragLocator,  /* fragment is in the data fork */
        kZeroOffset,            /* beginning offset of fragment */
        kCFragGoesToEOF,        /* ending offset of fragment */
        "mooApp"                /* name of the fragment*/
    }
```

- The value `kPowerPCCFragArch` indicates that this fragment was created for use with the PowerPC runtime environment.
- The value `kIsCompleteCFrag` indicates that the fragment is complete by itself.
- The constant `kNoVersionNum` in the next two fields has the value 0, a valid version number.

- The constant `kDefaultStackSize` in the next field indicates that the stack should be given the default size for the current software and hardware configuration. In System 7, you can use stack-adjusting techniques that call `GetApplLimit` and `SetApplLimit` if you determine at runtime that your application needs a larger or smaller stack.
- The constant `kNoAppSubFolder` indicates that there is no library search folder.
- The value `kApplicationCFrag` indicates that this is an application.
- The value `kDataForkCFragLocator` indicates that the fragment is stored in the data fork of the file.
- The values `kZeroOffset` and `kCFragGoesToEOF` in the next two fields indicate that the container for the fragment starts at the beginning of the data fork and ends at the end of the data fork.
- The default fragment name is usually the name of the output file from the linker, but you can assign a specific name if you wish.

A CFM-68K Application 'cfrg' 0 Resource Definition

Listing 1-8 shows a sample `'cfrg'0` resource definition for a CFM-68K runtime application. The fields that have values different from those in a PowerPC application `'cfrg'0` resource entry are underlined.

Listing 1-8 A sample `'cfrg'0` resource for a CFM-68K runtime application

```
#include "CodeFragmentTypes.r"
resource 'cfrg' (0) {
    {
        kMotorola68KCFragArch,  /* runtime environment */
        kIsCompleteCFrag,       /* base-level library */
        kNoVersionNum,          /* no current version number*/
        kNoVersionNum,          /* no oldest definition version number */
        kDefaultStackSize,      /* use default stack size */
        kNoAppSubFolder,        /* no library directory */
        kApplicationCFrag,      /* fragment is an application */
        kResourceCFragLocator,  /* fragment is in a resource */
        kRSEG,                  /* resource type = 'rseg' */
        kSegIDZero,             /* resource ID = 0 */
        "mooApp"                /* name of the application fragment*/
    }
```

- The constant `kMotorola68KCFragArch` in the first field indicates that this fragment was created for use with the CFM-68K runtime environment.
- The next underlined value, `kResourceCFragLocator`, indicates that this is a segmented application stored in resources.
- The next two underlined fields, `kRSEG` and `kSegIDZero`, tell the Code Fragment Manager that the initial container to load is contained in a resource of type `'rseg'` with a resource ID 0.

For more information about the structure of CFM-68K applications, see "CFM-68K Application Structure," beginning on page 9-3.

A Shared Library 'cfrg' 0 Resource Definition

Shared libraries have essentially the same `'cfrg'0` resource entry for both PowerPC and CFM-68K implementations (only the field indicating the runtime environment differs).

Listing 1-9 shows the `'cfrg'0` resource for an import library (plug-ins are identical except the fragment type is set to `kDropInAdditionCFrag`). Values that differ from an application's `'cfrg'0` resource are underlined.

Listing 1-9 A sample `'cfrg'0` resource for an import library

```
#include "CodeFragmentTypes.r"
resource 'cfrg' (0) {
    {
        kPowerPCCFragArch,      /* runtime environment */
        kIsCompleteCFrag,       /* base-level library */
        6,                      /* current version number*/
        4,                      /* oldest definition version number */
        kDefaultStackSize,      /* use default stack size */
        kNoAppSubFolder,        /* no library directory */
        kImportLibraryCFrag,    /* fragment is a library */
        kDataForkCFragLocator,  /* fragment is in the data fork */
        kZeroOffset,            /* fragment starts at offset 0 */
        kCFragGoesToEOF,        /* fragment occupies entire fork */
        "mooLib"                /* name of the library fragment */
    }
```

- The first two underlined fields store the current and definition version numbers that the Code Fragment Manager relies on for checking compatibility with client fragments. If you do not specify version numbers when you link, the version numbers are set to 0. See "Checking for Compatible Import Libraries," beginning on page 1-19, for more details.
- The application stack size field is ignored for shared libraries.
- The library directory field is ignored for shared libraries.
- The value `kImportLibraryCFrag` specifies that this is an import library. A plug-in would have the value `kDropInAdditionCFrag`.
- As you do with an application, you may supply a specific library name. However, for an import library you must do so before linking to a client because the fragment name is bound to the client at link time.

Special Symbols

A fragment can define three special symbols that are separate from the list of symbols exported by the fragment:

- a main symbol
- an initialization function
- a termination routine

The Main Symbol

The Code Fragment Manager returns the main symbol of a root fragment when preparing a closure; main symbols of any import libraries are ignored. The use of a fragment's **main symbol** depends upon the type of fragment containing it. For applications, the main symbol refers to the main routine, which is simply the usual entry point. The main routine typically performs any necessary application initialization not already performed by the initialization function and then jumps into the application's main event loop.

Applications must define a main symbol that is the application's entry point. Import libraries and plug-ins are not required to have a main symbol. However, plug-ins having a single entry point can use a main symbol instead of an exported symbol to avoid having to standardize on a particular name.

Note
In fact, the main symbol exported by a fragment does not have to refer to a routine at all; it can refer instead to a block of data. See "Using the Main Symbol as a Data Structure" (page 3-24) for more information. ◆

The Initialization Function

A fragment's **initialization function** is called as part of the Code Fragment Manager's fragment preparation process. You can use the initialization function to perform any actions that should be performed before any of the fragment's other code or static data is accessed. For example, in System 7, you often have to initialize various system services before you can use them (`InitWindows` for example). To make sure that all the required services are initialized before they are needed, you can put the calls to these services in an initialization function.

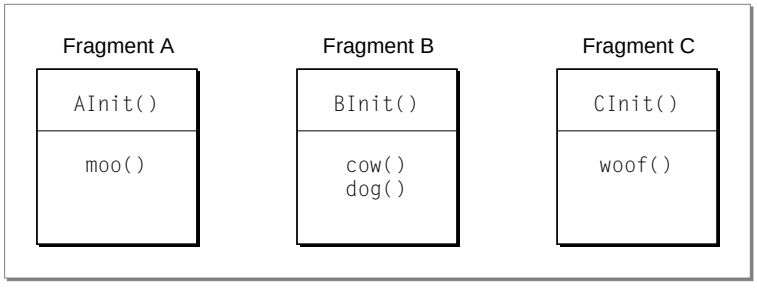
When a fragment's initialization function is executed, it is passed a pointer to a fragment initialization block, a data structure that contains information about the fragment. In particular, the initialization block contains information about the location of the fragment's container. For example, if an import library's code fragment is contained in some file's data fork, you can use that information to find the file's resource fork.

IMPORTANT
The initialization function must return a value. If an initialization function returns a nonzero value, preparation of the associated closure also fails. ▲

It's important to know when the initialization function for a fragment is executed. A good rule of thumb to remember is that a fragment's initialization function is executed whenever a new data section is instantiated for that fragment.

If the preparation of a fragment causes a (currently unprepared) import library to be prepared in order to resolve imports in the first fragment, the initialization function of the import library is executed before that of the first fragment. This makes sense because the initialization routine of the first fragment might need to use code or data in the import library. For example, Figure 1-12 shows three fragments and their initialization functions.

Figure 1-12 Three fragments with initialization functions



If fragment A imports symbols from fragment B, and fragment B imports symbols from fragment C, then C's initialization function must be run first, followed by B's, and then A's.

If you have two import libraries that depend upon each other, you may specify during the build process which should be initialized first.

Note you can run into problems if the initialization function of the import library requires routines that must be imported from another fragment. For example, in Figure 1-12, if the initialization function `AInit` imports `cow` from library B, and library B's routine `dog` imports `woof` from library C, you cannot guarantee that library C is initialized before it is needed. In general, your initialization function should be kept simple by avoiding accessing imported symbols.

The Termination Routine

A fragment's **termination routine** is executed only when a fragment's data instantiation is released. For example, if a fragment's data is globally shared between two applications, the fragment's termination routine would not be executed until both applications have quit. Note there is no guarantee that the termination routine will be run if your application crashes or otherwise terminates unnaturally.

You can use the termination routine to undo the actions of the initialization function or perform simple cleanup operations to preserve data (such as flushing file buffers). To avoid problems with circular dependencies, your termination routine should not reference symbols from other fragments.

When a process quits, the closures associated with the process are released in a first-in/first-out manner. That is, the first closure prepared is the first released.

This generally ensures that the Code Fragment Manager does not release a connection that another closure may depend upon. For example, if a process contains an application that prepared a plug-in, when the application quits, the application's closure is released first.

In general, your termination routine should be as simple as possible. For example, you may have your termination routine flush internal I/O buffers to any open files, but you don't need to actually close the files since the process termination sequence takes care of this action.

CHAPTER 2

Indirect Addressing in the CFM-Based Architecture

Contents

This chapter discusses the indirect addressing model used in the CFM-based runtime architecture. This material is presented separately as it does not relate directly to the Code Fragment Manager.

The overview section describes the fundamental concepts underlying the indirect addressing model. Everyone who is writing programs for the CFM-based architecture should read this section. In addition, this chapter provides details of how the indirect addressing model is implemented for the PowerPC and 68K Mac OS platforms. If you are writing a program that requires such low-level details (a compiler, for example) you should read these sections after the overview.

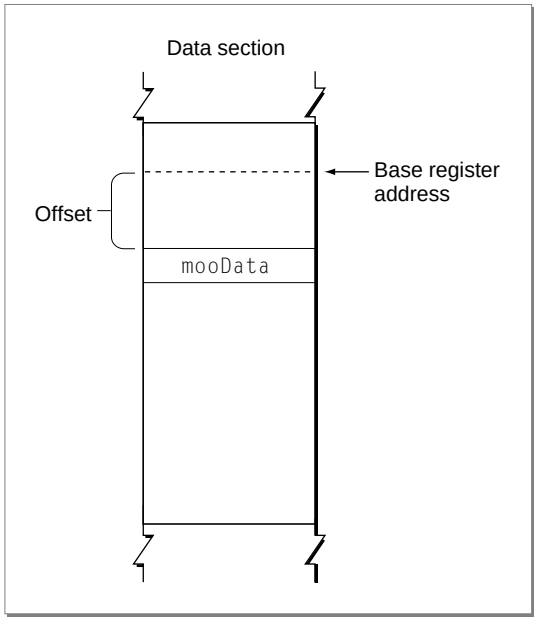
This chapter assumes knowledge of terms and concepts introduced in Chapter 1, "CFM-Based Runtime Architecture." In addition, read Chapter 3, "Programming for the CFM-Based Runtime Architecture," if you are planning to write CFM architecture–based programs.

Overview

Two methods exist for addressing data: direct addressing and indirect addressing. The choice of addressing method for any particular data item is determined by the compiler. Direct addressing is accomplished by using a **base register** to access an area of memory called the **direct data area.** Direct data items can be referenced as an offset from the address stored in the base register. Figure 2-1 shows an example of direct addressing.

Note
The term *direct addressing* as used in this chapter actually assumes one level of indirection (using the base register) and is therefore not the same as *absolute addressing* in assembly-language terminology. Similarly, *indirect addressing* actually possesses two traditional levels of indirection. ◆

Figure 2-1 Direct addressing of data

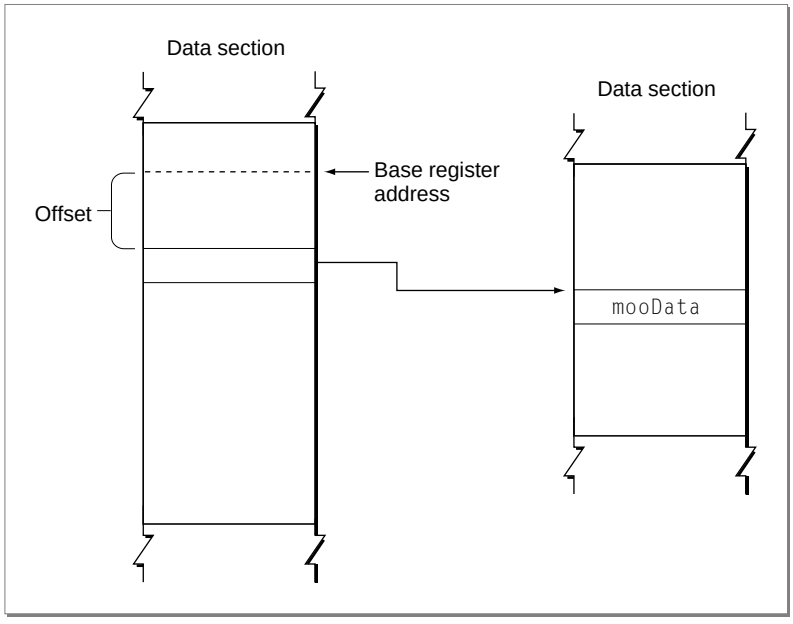


Direct addressing is simple and efficient, but since the offset bits in a given instruction can address only a certain amount of memory (typically ±32 KB), space limitations can occur if you have large data items or many data items.

If you are writing a compiler, you should store as many items as possible in the direct data area because this reduces access time. Small data items (that is, equal to or smaller than pointers) should always be placed in the direct data area.

The alternative is indirect addressing, where the item in the direct data area is not the data itself, but a pointer to that data. Since you are no longer restricted by addressing limitations, you can access large data structures. Figure 2-2 shows an example of indirect addressing.

Figure 2-2 Indirect addressing of data



The additional advantage of accessing symbols indirectly through pointers is that the symbols being referenced do not need to be present at build time. The components that make up a functional program can be stored separately if you can fix up the pointers to point to the correct symbols at runtime. In the CFM-based architecture, indirect addressing makes the use of imported and exported symbols possible.

Before preparation by the Code Fragment Manager, a fragment contains only a reference for each imported symbol. During the fragment preparation process, the Code Fragment Manager resolves all these references by searching for the code and data they refer to and replacing the references with relevant addresses.

Indirect addressing also provides the following benefits:

- Symbols external to a fragment can be specified by name, not by address. This allows the symbols to be grouped into import libraries.
- Data can be specified by name, not by address.
- Callback routines can be specified by name, not by address.

- Using the base register allows multiple connections with independent data sections in the same address space. For example, in System 7, all applications share the same address space, so allowing a fragment to have multiple connections in that space makes it possible to have shared libraries.
- An import library can have multiple connections associated with it, each linked to a different application.

Indirect addressing of data items is simple. Knowing the address stored in the base register and the offset into the direct data area, you can access the pointer to the data and consequently the data itself. For example, to find the proper address of an imported data item, a fragment adds the offset of the pointer to the import within the direct data area (determined at compile time) to the value stored in the base register. The result is the address of a pointer to the data item.

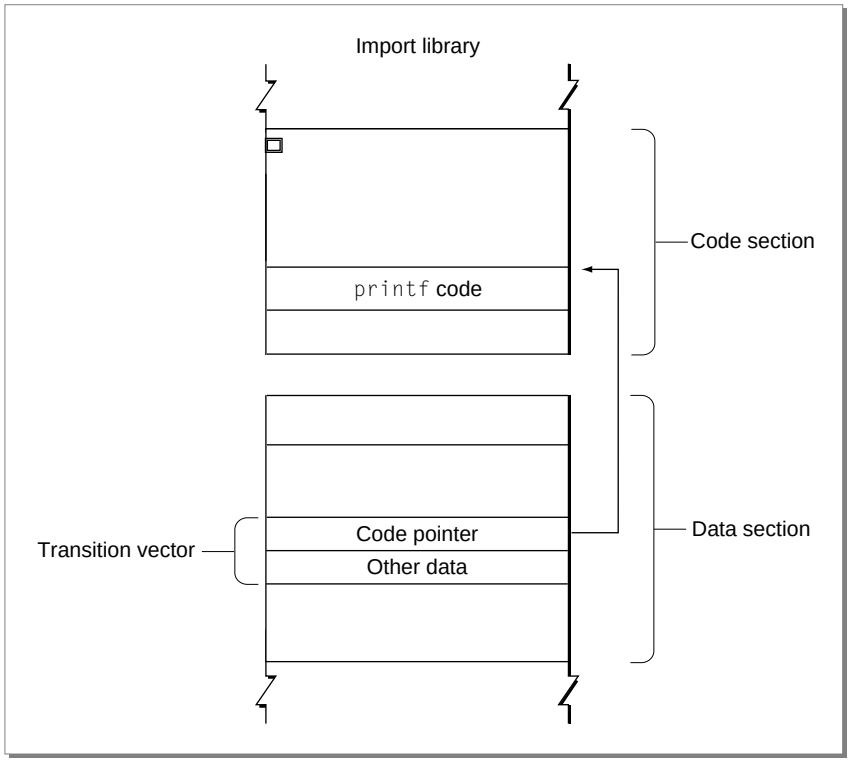
Note
The same indirect method is used to access global variables; the pointer merely points to the current fragment rather than a different one. ◆

Indirect addressing of routines is a little more complicated, but it is essentially similar. Indirect calls to routines must pass through the routine's **transition vector.** A transition vector is a data structure in the called fragment's data section whose first element is the address of the routine to be called. Any pointer to a routine (such as those used by C++ virtual method calls) actually points to the routine's transition vector, whether or not the routine is in the same fragment.

Indirect calls branch to the routine address (the first element of the transition vector) and store the address of the transition vector in a specific register (the details vary depending on the platform). This allows the called routine to access other elements in the transition vector (if any). The generated code usually also varies slightly for named calls (such as calls to imported routines) versus pointer-based calls (C function pointers or C++ virtual functions, for example).

The basic structure of a transition vector is shown in Figure 2-3.

Figure 2-3 A transition vector



A routine's transition vector is accessed through the base register, just like any other piece of data. As with other data, it is generally more efficient to place the transition vector in the direct data area. Control can then pass from the transition vector to the called routine.

The transition vector can contain any number of elements in addition to the routine address. These other elements may be used by the called routine in any way useful. For example, the PowerPC and CFM-68K implementations typically store a pointer to the called fragment's direct data area in a routine's transition vector; this method of storing the pointer allows the called routine to access its own data.

PowerPC Implementation

The PowerPC implementation of the indirect addressing model is fairly straightforward. The PowerPC runtime environment uses general-purpose register GPR2 as the base register. Note that the use of this register to access the direct data area is a convention, not a requirement. Debuggers and other analytical applications should not assume that GPR2 is the base register.

Note
Historically (from IBM documentation) the set of pointers to a fragment's indirectly accessed data was referred to as the Table of Contents and its base register was called the Table of Contents Register (RTOC). ◆

To access imported data or indirect global data, the build-time offset of the global data item is added to the value in GPR2. The result is the address of a pointer that points to the desired data.

To access imported routines, the offset of the routine is added to the value in GPR2, as in the data version, but the result points not directly to the routine, but to a transition vector.

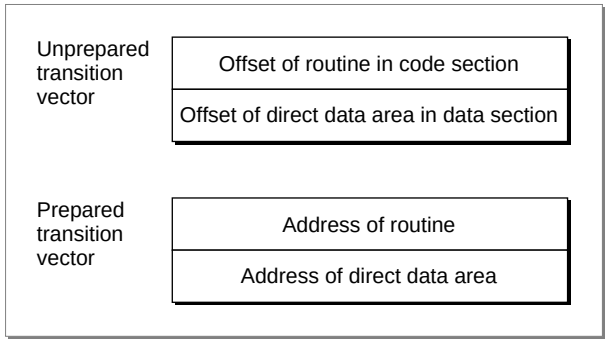
The PowerPC transition vector typically contains two elements. The first must be the address of the routine being called. By convention the second element contains the address of the called fragment's direct data area.

Prior to preparation, the transition vector contains

- the offset of the routine being called from the beginning of the code section
- the offset of the direct data area from the beginning of its data section

During preparation, the Code Fragment Manager adds the code and data section start addresses to the offset values, generating absolute addresses for the routine and the location of the direct data area. Figure 2-4 shows the unprepared and prepared versions of the transition vector.

Figure 2-4 Unprepared and prepared PowerPC transition vectors



Note
The transition vector may contain any number of 4-byte fields. Currently only the first two are used. During an indirect call, GPR12 is assumed to point to the transition vector itself; this convention allows the called routine to access any additional fields in the transition vector beyond the first two. ◆

In order for indirect calls to work properly, certain requirements must be met on the part of the calling routine and the called routine. These requirements are as follows:

- For each routine call, the compiler generates a PC-relative branch followed by an instruction to restore GPR2.
- When entering the called routine, GPR12 points to the transition vector and GPR2 contains the second word of the transition vector.
- When returning to the calling routine, the old GPR2 value resides on the stack at `20(SP)` (in the linkage area).

How these requirements are implemented is determined by convention. For example, in the PowerPC runtime environment, glue code in the calling routine handles loading the proper values into GPR2 and GPR12. Any other actions are also determined by convention.

Glue Code for Named Indirect Calls

If the routine call is referenced indirectly by name, the linker generates glue code in the calling fragment and directs the compiler-generated branch to this code. If the linker finds that a call is local (that is, not cross-fragment), it replaces the GPR2 restore instruction with a NOP instruction.

The glue code takes the following steps to switch the direct data areas and execute the actual called routine:

1. Loads the pointer for the transition vector into GPR12.
2. Saves the current value of GPR2.
3. Loads the next 4 bytes of the transition vector into GPR2 (effectively switching the direct data area).
4. Jumps to the start of the actual routine.

Upon return, control passes directly to the caller (not the glue code) which restores the saved value of GPR2.

Listing 2-1 shows some sample glue code.

Listing 2-1 Glue code for a cross-fragment call

```
    bl  moo_glue                    ; call the cross-fragment glue
    lwz     R2, R2_save_offs(SP)    ; restore the caller's base pointer

    ...

moo_glue:
    lwz     R12, tvect_of_moo(R2)   ; get pointer to moo's transition
                                    ;  vector
    stw     R2, R2_save_offs(SP)    ; save the caller's base pointer
    lwz     R0, 0(R12)              ; get moo's entry point
    lwz     R2, 4(R12)              ; load moo's base pointer
    mtctr   R0                      ; move entry point to Count Register
    bctr                            ; and jump to moo
```

Note
The linker generates custom glue for each routine since the glue code contains the direct data area offset of the routine's transition vector. ◆

Glue Code for Pointer-Based Calls

Pointer-based function calls must make use of the transition vector since the eventual call may be cross-fragment. (A routine pointer does not point to the code, but to the transition vector of the called routine.)

The glue code for a pointer-based call, as shown in Listing 2-2, is very similar to that for the named indirect call.

Listing 2-2 Glue code for a pointer-based call

```
    lwz R12, address_of_TVector
    bl      ptr_glue
    lwz     R2, R2_save_offs(SP)    ; restore the caller's base pointer

    ...

ptr_glue:
    lwz     R0, 0(R12)              ; get the entry point
    stw     R2, R2_save_offs(SP)    ; save the caller's base pointer
    mtctr   R0                      ; move entry point to Count Register
    lwz     R2, 4(R12)              ; load the new base pointer
    bctr                            ; jump through the Count Register
```

CFM-68K Implementation

In the CFM-68K runtime environment, the A5 register acts as the base register.

To access global, static, or imported data, the build-time offset of the direct data item is added to the value in A5. The result is the address of a pointer that points to the desired data.

To access imported routines, the offset of the routine is added to the value in A5, as in the data version, but the address at that location points not directly to the routine, but to a transition vector.

Note
CFM-68K transition vectors reside above the jump table in the direct data area of the called routine. This positioning is dictated by segmentation requirements. ◆

A CFM-68K import library's transition vector is typically similar to that of a PowerPC import library, containing two 4-byte elements: the address of the routine being called and the address of the called fragment's direct data area (A5 world).

Note
Transition vectors in CFM-68K application fragments include a third field of segment information to allow them to properly address routines in a segmented application. For more information about the structure of CFM-68K applications and the application launch process, see "CFM-68K Application Structure," beginning on page 9-3. ◆

Direct and Indirect Calls

A direct call does not require switching of the direct data area since it makes use of the calling party's direct data area by default. A direct assembly-language call to the function `mooFunc` would simply be

```
BSR.L               _$mooFunc
```

where `_$mooFunc` signifies the internal entry point of `mooFunc`.

IMPORTANT
When discussing routine calls and the direct data area switching method, the **external entry point** refers to the entry point of the routine when called indirectly through a transition vector. A direct (in-fragment) call enters a routine through the **internal entry point.** ▲

To allow the switching of the direct data area, the CFM-68K runtime architecture specifies that a procedure pointer points to a transition vector.

All indirect or cross-fragment calls go through a transition vector. The fragment uses the code and data world pair to set up a direct data area associated with the called routine. An indirect call to the function `mooFunc` would be as follows:

```
MOVE.L              _@mooFunc,A1    ; load transition vector into A1
MOVE.L              (A1)+, A0       ; get code address
JSR                 (A0)            ; make the call
MOVE.L              "lcl", A5       ; restore A5 from "cl"
                                    ; after the call.
```

The term "`lcl`", which can be either a memory-based load variable or a register variable, is the location where the procedure saved its own A5 value prior to the call.

Note
Unlike the PowerPC case, glue code in the *called routine* is responsible for switching the direct data area in CFM-68K. However, the other actions of the caller's code (loading the transition vector into a register, calling the routine, and restoring the base register after the call) are identical. ◆

The Direct Data Area Switching Method

In the CFM-68K runtime environment, the standard direct data area switching procedure takes the following steps.

1. The program uses the transition vector to jump to the external entry point of the procedure. At this point, the A1 register points to the second word of the transition vector, which contains the address of the direct data area.
2. The external entry point loads the A5 register with the new direct data area address (using the register A1) and then enters the internal entry point.
3. The function's prolog code is executed, part of which saves a copy of A5 in case the function must in turn make other indirect or cross-fragment calls.
4. The program executes the function. If the routine makes any indirect or cross-fragment calls, it restores the saved value of A5 after each such call.
5. After executing the function, the program then runs the epilog and throws away its local variables (including the saved copy of A5).
6. After running the epilog, the program returns to the calling fragment.

Direct callers and indirect callers can enter the procedure at different locations, so you can set up slightly different prolog sequences depending on the type of call.

Listing 2-3 illustrates glue code surrounding a simple function call.

Listing 2-3 Glue code for a simple function

```
MOVE.L              (A1), A5                ; set up A5 from A1
LINK                A6, #LOCALS             ; (this is the internal entry point)
MOVEM.L             <REGSET/A5>,-(A7)       ; save new A5

<body of function  here>

MOVEM.L             (A7)+, <REGSET>         ; note A5 not restored here
UNLK                A6
RTD                 #PARAM_CT
```

If the function itself makes indirect or cross-fragment calls, you must save the A5 value before the call and restore it after each return. Listing 2-4 shows how to handle an indirect call within an indirectly called function:

Listing 2-4 Making an indirect call from within an indirectly called function

```
MOVE.L              (A1), A5                ; set up A5 from A1
LINK                A6, #LOCALS             ; the reserved space
MOVEM.L             D7/D6/A5, -(A7)         ; save new A5 at -12(A6)

...

                                            ; now making cross-fragment call to
                                            ; the imported function mooCall
MOVE.L              _@mooCall(A5), A1       ; load transition vector into A1 via
                                            ;  the pointer to the transition vector
MOVE.L              (A1)+,A0                ; get code address
JSR                 (A0)                    ; call function
MOVE.L              -12(A6), A5             ; restore A5 from saved location
```

```
...

                                        ; no A5 restore here (but pop saved
MOVEM.L         (A7)+, D7/D6            ; data registers)
UNLK            A6                      ; UNLK compensates for unbalanced stack
RTD             #PARAM_CT
```

Note
You do not have to save your A5 value on the stack. In some cases (such as when you know the called procedure will make a lot of indirect calls) it may be advantageous to save your A5 value in a data register, or even another address register. ◆

In certain cases, you may omit some of the switching steps to optimize your code. Three different optimization possibilities exist:

- You can choose to not save A5 in the prolog. This choice is useful only when you are certain that the routine you are calling will never make any indirect or cross-fragment calls and thus will never need to restore A5. Routines using this optimization may still use A5, however.
- You can remove the external entry point and transition vector. This choice removes the initial `MOVE.L (A1), A5` instruction and is equivalent to tagging the routine as `internal` during a compile. You should use this optimization only if you are sure that the routine will never be called indirectly or from another fragment.
- You can remove the `MOVE.L (A1), A5` instruction but keep the transition vector. This optimization works only for calling routines that never use the A5 register during execution (for example, a leaf routine that doesn't access global variables). Note that all the following actions do use A5 and disqualify routines from using this option:
  - Direct (in-fragment) calls, because the called procedure may use A5, or the call may have to go through the jump table (which uses A5). Note that segmented shared libraries use the jump table if calling direct between two code segments.
  - Any cross-fragment call or access to an imported data item, because the CFM-68K code uses the A5 register to access such data items indirectly.

Programming for the CFM-Based Runtime Architecture

Contents

This chapter contains practical information about programming for the CFM-based runtime architecture, including guidelines for building shared libraries. Note that the topics in this chapter are independent of one another and do not need to be read in any particular order.

This chapter assumes familarity with the terms and concepts presented in Chapter 1, "CFM-Based Runtime Architecture," and Chapter 2, "Indirect Addressing in the CFM-Based Architecture."

Calling the Code Fragment Manager

If your application uses plug-ins, you must prepare them explicitly by calling the Code Fragment Manager from your code. This section describes some of the routines you can use to prepare fragments and find symbols exported from other fragments.

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 plug-ins. ▲

Preparing Code Fragments

If the fragment is an import library that contains a `'cfrg'0` resource, you can use the Code Fragment Manager's `GetSharedLibrary` function to prepare the fragment. If the fragment is stored in a disk file, you call the `GetDiskFragment` function. If the fragment is stored in a resource, you need to place the resource into memory (using normal Resource Manager and Memory Manager routines) and then call the `GetMemFragment` function. In general, however, you should avoid storing fragments in resources. Resource-based fragments do not gain the benefits of file-based fragments (such as file mapping directly from the file's data fork), so you should use them only when you have no other choice.

For complete information about the Code Fragment Manager routines, see *Inside Macintosh: PowerPC System Software*. The APIs defined in that book apply for both the PowerPC and CFM-68K implementations.

In general, the overhead involved in preparing the code fragment and later releasing it is not trivial, so you should avoid closing the connection to a prepared fragment (that is, calling `CloseConnection`) until you are finished using it.

IMPORTANT

When called to prepare a plug-in, the Code Fragment Manager automatically prepares all the fragments that make up the plug-in's closure. That is, if the plug-in imports symbols from an import library, that library is also prepared; you do not have to explicitly prepare the library. ▲

Listing 3-1 shows how to prepare a fragment using `GetSharedLibrary`.

Listing 3-1 Preparing a fragment using `GetSharedLibrary`

```
myErr = GetSharedLibrary(myLibName, KPowerPCCFragArch, kPrivateCFragCopy,
                         &myConnID, (Ptr*)&myMainAddr, myErrName);
if (myErr) {
    AlertUser(myErr);
}
```

The fragment name is held in `myLibName` and it is specified to be a PowerPC fragment. The Code Fragment Manager follows its standard search path to find the library. See "Searching for Import Libraries," beginning on page 1-16, for more information on the search procedure.

Note that the preparation fails if the preparation of any of the fragments that make up the closure fails. The error term `myErrName` then contains the name of the fragment that caused the failure.

Listing 3-2 show how to prepare a disk-based fragment.

Listing 3-2 Preparing a disk-based fragment

```
myErr = GetDiskFragment(&myFSSpec, 0, kCFragGoesToEOF, myToolName,
                    kPrivateCFragCopy, &myConnID, (Ptr*)&myMainAddr,
                    myErrName);
if (myErr) {
    AlertUser(myErr);
}
```

Listing 3-3 shows how to prepare a resource-based fragment.

Listing 3-3 Preparing 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, kPrivateCFragCopy, &myConnID,
                    (Ptr*)&myMainAddr, myErrName);
if (myErr) {
    AlertUser(myErr);
}
```

The code in Listing 3-3 places 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 memory address of the fragment. Because `GetResource` returns a handle to the resource data, Listing 3-3 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 `kPrivateCFragCopy` passed as the fourth parameter requests that the Code Fragment Manager allocate a new copy of the fragment's global data section.

Like other fragments a resource-based fragment must remain locked in memory and has separate code and data sections. You have access to the connection ID of the resource-based fragment, so you can call Code Fragment Manager routines like `CloseConnection` and `FindSymbol`.

Note
Some PowerPC executable resources are specially written to model a classic 68K stand-alone code resource. These accelerated resources do not have all the freedom of a true fragment. See "Accelerated and Fat Resources," beginning on page 7-4, for information about how to write and call an accelerated resource. ◆

Releasing Fragments

To programmatically release fragments from memory, you use the `CloseConnection` routine. A call to `CloseConnection` is simply

```
myErr = CloseConnection(&myID);
```

where `myID` is the ID value received when you called the Code Fragment Manager to prepare the fragment. Note that you cannot call `CloseConnection` using the ID value received when using the `FindCFrag` option or the ID passed by a fragment's initialization block (when executing an initialization function).

The `CloseConnection` routine actually releases the closure associated with the ID and decrements the associated reference counts. If any reference counts drop to 0, the Code Fragment Manager releases the associated section, connection, or fragment container.

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

Getting Information About Exported Symbols

In cases in which you prepare 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 `FindSymbol`, `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 0 to one less than the total number of exported 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 exported symbols in a fragment. For example, the code in Listing 3-4 prints the names of all the exported symbols in a particular fragment.

Listing 3-4 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 = 0; 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 *Inside Macintosh: PowerPC System Software* for details.

Using Shadow Libraries

In some cases you might want to prepare import libraries on an “on-call” basis the same way you would with plug-ins. For example, if you only occasionally

use routines from `mooLib` in your application, you may not want to take up excess memory when `mooLib` is not required. In such cases, you should create a shadow library. A **shadow library** is essentially a small library whose only purpose is to prepare an import library when a symbol from that library is required.

For example, suppose you have a simple header file with this function:

```
StatusType FunctionOne (ParamOneAType param1A, ParamOneBType param1B);
```

Listing 3-5 shows how to access these functions through a simple shadow library.

Listing 3-5 Sample code found in a shadow library

```
/* Function pointers used internally by the shadows. */
typedef StatusType (* FunctionOnePtr) (PramOneAType param1A, ParamOneBType param1B);

/* The initial version of the function pointers. */
static StatusType PrepareAndCallF1 (ParamOneAType param1A, ParamOneBType param1B);
static FunctionOnePtr gFunctionOne = PrepareAndCallF1;

/* The initial version of the function pointer, which does setup first, and then */
/* calls through to the actual function. */

static StatusType PrepareAndCallF1 (ParamOneAType param1A, ParamOneBType param1B)
{
    OSErr err = Setup ();
    if (err == noErr)
        err = (*gFunctionOne) (param1A, param1B);
    return err;
}

/* The shadow implememtation of FunctionOne, which calls through the */
/* function pointer, which itself could point to the setup version */
/* (first time) or to the actual routine (second time or later). */

StatusType FunctionOne (ParamOneAType param1A, ParamOneBType param1B)
    {
    return (*gFunctionOne) (param1A, param1B);
    }
```

```
static OSErr Setup (void)
    {
    CFragConnectionID connID;
    OSErr err = GetSharedLibrary ("\pRealImplementation",
                                  kCompiledCFragArch, kReferenceCFrag,
                                  &connID, NULL, NULL);

    if (err == noErr)
        {
        FunctionOnePtr p1;
        err = FindSymbol (connID, "\pFunctionOne", (Ptr *) & p1, NULL);
        if (err == noErr)
            gFunctionOne = p1;
        }
    return err;
    }
```

This example just uses local pointers to go to the "right" function. The first time through, these pointers take you to routines that call the Code Fragment Manager to prepare the real library. Subsequent times they take you to the real routines. This implementation requires no changes to clients (for example, you can change the import library without recompiling or relinking the clients).

Note that this example works only if all functions in the library return some sort of success/failure indication. This could be through an explicit status value, a null/nonnull pointer, and so on. Also, this example is not preemptive thread safe, and it does not have sophisticated error checking. If you are writing code to prepare a shadow library, you should anticipate errors such as the following:

- The initialization function in the library fails.
- The Code Fragment Manager cannot find a compatible library.
- The Code Fragment Manager cannot find the required symbols in the library.

Also, if your shadow library may be released at some point, you should include code in the library's termination routine to release any libraries it has prepared and to perform any other necessary cleanup.

Requirements for Executing CFM-68K Runtime Programs

To run CFM-68K runtime programs, target computers must have the following configuration:

- System software version 7.1 or later.
- A 68020 or later microprocessor.
- The CFM-68K runtime library, which contains the Code Fragment Manager and shared library routines that the CFM-68K program accesses during execution. This library may be available as part of the system software or as a system extension called the CFM-68K Runtime Enabler.

If the Code Fragment Manager is not present when an attempt to launch a CFM-68K application is made, a message indicates that the CFM-68K Runtime Enabler is needed.

If you want to check on the availability of the Code Fragment Manager from classic 68K code, you can call the `Gestalt` function with the selector `gestaltCFMAttr` in a routine similar to the following:

```
Boolean HaveCFM()
{
    long response;
    return ( (Gestalt(gestaltCFMAttr, &response) == noErr) &&
                (((response >> gestaltCFMPresent) & 1) != 0));
}
```

For more information about `Gestalt` and the Gestalt Manager, see *Inside Macintosh: Operating System Utilities.*

CFM-68K programs run transparently side by side with classic 68K applications. The Process Manager reads the `'cfrg'0` resource at application launch time. The `'cfrg'0` resource tells the Process Manager whether the application contains CFM-68K runtime code and, if so, where that code is located. If the Process Manager cannot find a `'cfrg'0` resource, it assumes that the application is a classic 68K application, where the executable code is contained within `'CODE'` resources in the application's resource fork. For more details of the CFM-68K application launch process, see Chapter 9, "CFM-68K Application and Shared Library Structure."

If the target 68K computer does not support file-mapping, it must have enough RAM installed to load all the shared libraries required by the CFM-68K program. At least 8 MB of RAM is suggested for target computers.

Using Stub Libraries at Build Time

Stub libraries are import libraries that export symbols but do not contain any code. Instead of linking against fully functional import libraries, you can link against a stub library, since all you need at build time is the definition of the library's API.

Stub libraries are also useful when you have a circular dependency between import libraries. For example, if the library `mooLib` imports symbols from `cowLib` and `cowLib` imports symbols from `mooLib`, then a problem arises: you cannot build `mooLib` without linking with `cowLib` and you cannot build `cowLib` without linking to `mooLib`. The solution is to begin by linking against a stub version of one library. You can build `mooLib` by linking to a stub of `cowLib` (which allows you to resolve imports from `cowLib`), and then you can build the real `cowLib` by linking it to `mooLib`.

Weak Libraries and Symbols

During the build process, you can designate certain symbols or import libraries to be *weak* (usually with linker options), which indicates to the Code Fragment Manager that the symbol or library is not required for execution. For example, an application `mooProg` may designate the QuickTime shared library as a **weak library.** Then, while it can make use of QuickTime features if the library exists, it can still launch and execute normally without it. Similarly, a **weak symbol** is an imported symbol that does not have to be present at launch time. Weak symbols are sometimes called *soft imports.*

IMPORTANT
Although the Code Fragment Manager allows weak imports to remain unresolved at runtime, the application is still responsible for checking to see if the symbol or library was found and taking appropriate action. For example, if a library was not found, the application might display a message and set a flag to avoid accessing routines or data imported from that library. ▲

If the Code Fragment Manager cannot find imported symbols designated as weak, all references to these imports are replaced with the value `kUnresolvedSymbolAddress`. Listing 3-6 shows how you can check for weak imports using this value.

Listing 3-6 Testing for weak imports

```
extern int dogCow (char *, ...);
...
if (dogCow == kUnresolvedSymbolAddress)
    DebugStr("\dogCow is not available.");
else
    printf("Hi Clarus\n");
```

The Code Fragment Manager checks for weak libraries before doing any preparation (resolving symbols and so on), and if the library exists, it is subsequently handled as a normal import library. For example, if an error occurs during preparation of the library, the Code Fragment Manager may abort the launch procedure, even though the library was designated as weak.

▲ **WARNING**
You should not use the `Gestalt` function to check for weak imports or weak libraries. ▲

If the Code Fragment Manager cannot find a weak library, you cannot subsequently resolve symbols imported from that library by calling Code Fragment Manager routines (`GetSharedLibrary`, for example).

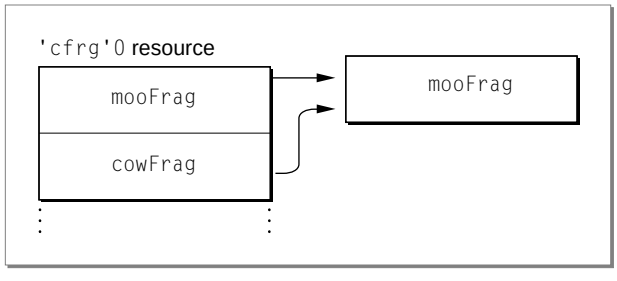
Multiple Names for the Same Fragment

The CFM-based architecture allows you to assign multiple names to a single fragment. For example, if you have a fragment that implements multiple SOM classes, you can assign a separate name for each class, all of which point to the same fragment.

You store multiple names as multiple `'cfrg'0` entries. As mentioned earlier, the `'cfrg'0` resource is actually an array, so you can store as many fragment descriptions as you like.

For example, the `'cfrg'0` resource in Figure 3-1 contains two fragment entries, `mooFrag` and `cowFrag`, which both point to the same fragment (that is, their `'cfrg'0` resource entries map to the same location). If the Code Fragment Manager is called to prepare `mooFrag` and then called sometime later to prepare `cowFrag`, it knows that they are the same fragment and treats them as such. For example, if the preparation request for `cowFrag` came from the same process, it increments the reference count for `mooFrag` and creates a closure using the existing connection. In this manner it is possible to create “aliases” for fragment names.

Figure 3-1 Two names for a single fragment



You can use aliasing to update older libraries without having to change the client fragments that import from them. For example, say you build a library `cowFrag` and create several applications that use it. Sometime later you build another library `mooFrag` that contains all the functionality of `cowFrag` as well as

some new features. If the 'cfrg'0 entry for mooFrag contains an entry for both mooFrag and cowFrag, then the following are possible:

- Applications built with mooFrag can run with mooFrag and use all of the available features.
- Applications built with cowFrag can run with mooFrag and use the features previously available in cowFrag.

Import Library Techniques

Sometimes when you modify an import library, the new version may not remain fully compatible with older versions. As a rule of thumb, the developer should think about compatibility issues for versions of your import libraries in the following cases:

- the API for the library changes
- the input or output behavior of any library routine changes

There are a number of ways to check or maintain compatibility between successive versions of an import library. Table 3-1 shows some methods for checking or maintaining compatibility. Each method has advantages and disadvantages, and some of them may be used in conjunction with each other.

Table 3-1 Methods for maintaining import library compatibility

Method	Advantages	Disadvantages
Use no version numbers and no weak symbols	No work required at build time.	Missing symbols cause program failure.
Declare weak symbols in client	Allows limited functionality with whatever symbols are available.	Code must check for presence of library exports. Difficult to keep track of all weak symbols.

continued

Table 3-1 Methods for maintaining import library compatibility (continued)

Method	Advantages	Disadvantages
Use PEF version numbering	The Code Fragment Manager automatically checks for compatibility.	Program fails if a compatible library is not found. Version numbers can't take into account all possible compatibility cases.
Change names for newer import libraries	Eliminates the need for version or symbol checking. Different versions of a library can appear in memory at the same time.	Only useful if the newer version cannot support older clients. Multiple names for libraries with similar functionality can be dangerous.
Create an alias library name using multiple `'cfrg'0` entries	Allows older clients to use a newer library with a different name.	Generally only useful when combining older implementations into one library.
Put new symbols in new logical libraries	The functionality of an import library never changes.	The number of import library names accumulates over time.
Use reexport libraries	Allows older clients to use multiple newer libraries in place of an older one.	Generally only useful when an older implementation splits into several newer libraries.

These methods are described more completely in the sections that follow.

Use No Version Numbers and No Weak Symbols

Choosing not to use any version numbers or weak symbols when developing import libraries provides rudimentary compatibility checking with no effort on the developer's part. That is, if a required symbol is not found, then the program preparation fails.

Note
Choosing no version numbers means that all three version numbers are set to zero. ◆

Declare Weak Symbols in Client

If symbols are added to a newer version of an import library, the developer can make sure that the newer clients can still link to the older library by declaring the new symbols to be weak. This method, however, has two drawbacks: the client application must check for all weak imports, and the developer must keep track of all weak exports.

For example, say you have a library `mooLib 1.0`, which contains the symbols `dog` and `cow`. Later you update `mooLib` to 2.0 by adding the symbols `woof` and `moof`. If `woof` and `moof` are declared weak by a client built with `mooLib 2.0`, the client can still run with `mooLib 1.0`; it just cannot use the symbols `woof` and `moof`.

The developer's code must check for the presence of all weak symbols before attempting to use them and perhaps inform the end user of any limited functionality if some symbols are not present. In addition, each time a new version of the library is created, the developer must create a new weak export list. For example, building clients with `mooLib 2.0` requires a weak export list of all symbols added after version 1.0. Building with version 3.0 would require a list of weak symbols added between 1.0 and 2.0 *and* a list of symbols added between 2.0 and 3.0. When building libraries for other developers, the developer would have to supply an updated export list for every version released.

Use PEF Version Numbering

The Code Fragment Manager relies on version numbers stored in the import library PEF containers to determine whether an implementation library is compatible with the definition stub library. The developer can assign version numbers as a redundancy check for possible library mismatches during a library's development. As shown in the example that follows, there are some cases that this method does not solve. This section gives several examples of when and how to change these version numbers when developing import libraries.

When using PEF versioning, the developer should use the following rules:

- The first library should have all three version numbers set to zero.
- The current version number can be incremented each time the developer releases a change to the library.
- The old definition version number should be incremented only if the developer changes the library's API in a manner that makes the library

incompatible with older clients (for example, removing a routine that older clients expect to see).

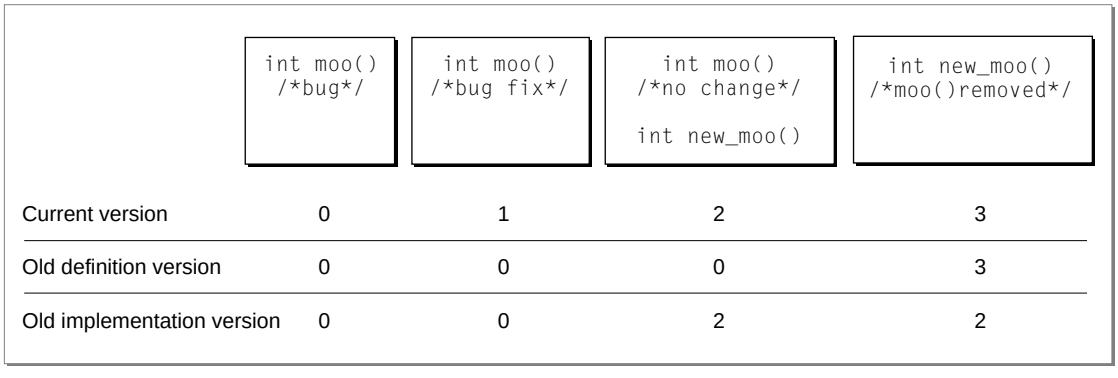
- The old implementation number should be incremented only if the developer makes additions to the API that new clients must depend on (for example, adding a routine that all new clients will require).

IMPORTANT

The version numbers encoded in an import library are for developer use only. The library compatibility version numbers do not need to correspond to the version numbers visible to the end user. ▲

Figure 3-2 shows the version numbers required for each update of a library `mooLib` that contains the function `moo`.

Figure 3-2 Changes to import library version numbers



Current version	0	1	2	3
Old definition version	0	0	0	3
Old implementation version	0	0	2	2

When you first build the library, you do not have to worry about compatibility, so all the version numbers are set to 0.

Now, suppose you find a minor bug in the function `moo` in your first version. After fixing the bug, you create version 1. Fragments built with version 0 can run on machines that contain version 1 without having to be updated, because the two versions of `moo` are compatible. Similarly, fragments built with version 1 can still run on machines that contain version 0 (even though it contains a bug). Therefore, the old implementation and old definition numbers remain at 0 while the current version number is raised to 1.

Now, suppose you update the library again to add a different implementation of function `moo`, called `new_moo`. The definition and implementation for function `moo` remain the same. This version becomes version 2 of the import library. Fragments built with either of the older versions still run on machines that have version 2 because they won't look for the function `new_moo`. However, fragments built with version 2 cannot run on machines containing older versions of the library because they cannot find an implementation for function `new_moo`. Therefore, version 2 of `mooLib` has an old *definition* version of 0 and an old *implementation* version of 2.

Finally, you remove function `moo`, so that only `new_moo` is supported, and build version 3 of the import library. Fragments built with older versions of the import library won't run with version 3 because they expect `moo` to be present. However, fragments built with version 3 run on machines that contain any version that has an implementation for `new_moo` (in this case, version 2 or version 3). Therefore, version 3 of `mooLib` should have a old definition version of 3 and an old implementation version of 2.

A drawback of simple PEF versioning is that if a compatible implementation library is not found, the program fails. Although this result is the same as using no versioning at all, PEF versioning can also prevent incompatible usage in cases where the symbols have not changed. In addition, PEF versioning acts as a redundancy check for possible library mismatches during development. Note that even when a compatible library is found, the client fragment cannot determine which version of the library was actually used.

In addition, each version number can represent only a single compatibility range. Depending on how the developer changes the library, it is possible to have pockets of compatibility appear in older versions that cannot be represented by the version numbers. As a trivial example, say you create a version 4 of `mooLib` that restores the function `moo`. Fragments built with version 4 cannot run with version 3 because version 3 does not contain `moo`; the old definition version number must be 4. However, this choice also disqualifies version 2, which does contain `moo` and would be a compatible library.

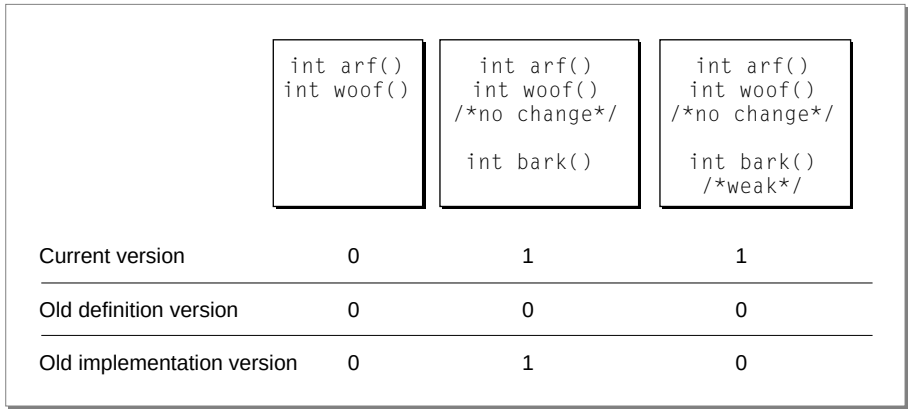
In some cases, the developer can increase the compatibility ranges by designating weak symbols in addition to PEF versioning. For example, say you have a library `dogLib` with the functions `woof` and `arf`. Normally, if you add a new function `bark` to `dogLib`, you must increase the current and old implementation version numbers as in the previous example. However, if the fragment that imports from `dogLib` declares `bark` to be weak, you have a little

more flexibility. For example, if version 0 is the original `dogLib` and version 1 contains the `bark` function, the following are true:

- A client fragment built with version 0 of `dogLib` can run with either version 0 or version 1, since it neither knows about nor uses `bark`.
- A client fragment built with version 1 of `dogLib` can run with either version 0 or version 1 if the client declares `bark` to be weak. (Note that the client fragment must check for the presence of `bark` and use it only if it is available.)

Therefore, if `bark` is weak, version 1 of `dogLib` can have both its old definition and old implementation version numbers set to 0. Figure 3-3 shows the version numbers for both variations of `bark`.

Figure 3-3 Version numbering with weak imports



Change Names for Newer Import Libraries

If the new version of the import library cannot support older clients, it is essentially a new library, so the developer could give the new library a different name to eliminate the need for version or symbol checking. For example, you could call a revision to `mooLib` that is not compatible with older clients

mooLib_2.0. However, since the Code Fragment Manager considers libraries with different names to be totally separate libraries, it is possible to have several instantiations of a library present in memory at the same time.

▲ **WARNING**

By simply renaming an import library, it is possible for one program to end up trying to use two different versions of an import library. For example, say an application uses mooLib and uses a third-party library that also requires mooLib. If the third-party developer decides to upgrade to mooLib_2.0, the application may end up trying to use both mooLib and mooLib_2.0. Because of this danger, the developer should avoid simply renaming newer versions of import libraries. For a safer method, see "Put New Symbols in New Logical Libraries" (page 3-21). ▲

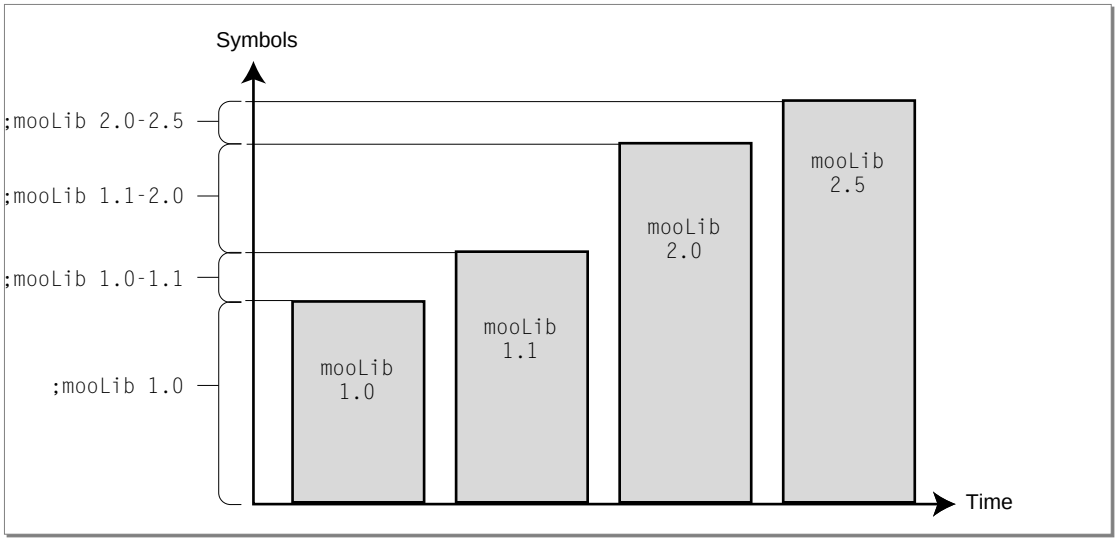
Create an Alias Library Name Using Multiple 'cfrg' 0 Entries

A developer can create aliases for library names by including additional 'cfrg'0 entries that point to the same fragment (see "Multiple Names for the Same Fragment" (page 3-13) for more information). This aliasing can be useful when combining several libraries into one fragment. For example, say you have libraries cowLib and dogLib that have been linked to a number of clients. You then decide to merge cowLib and dogLib into a new library dogCowLib. To ensure that clients originally built with cowLib or dogLib can still access those routines in dogCowLib, you must create separate 'cfrg'0 entries for cowLib and dogLib. These entries list the old fragment names but point to the container for dogCowLib.

Put New Symbols in New Logical Libraries

A developer can give logical names to different portions of a library and then have multiple `'cfrg'0` entries to point to a single implementation. For example, consider the breakdown of `mooLib` in Figure 3-4.

Figure 3-4 Multiple logical names for a single library



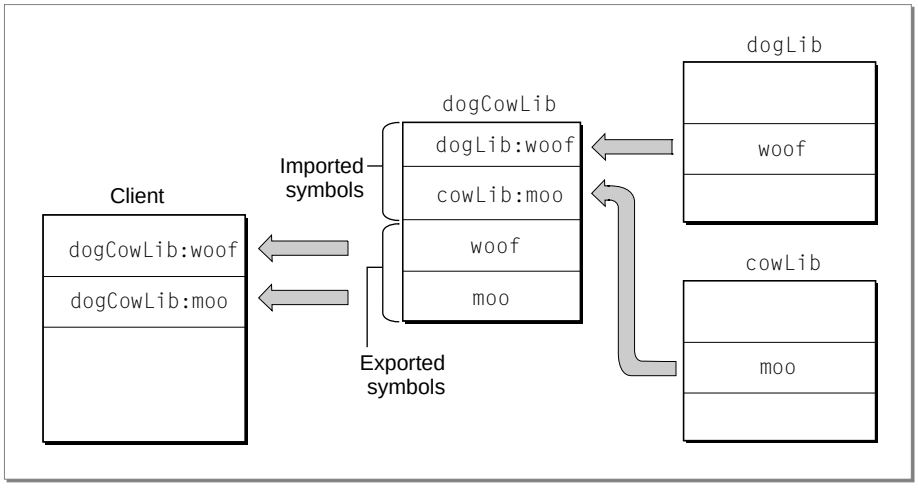
The updated portion of each new version has its own logical name. For example, if your program called a routine introduced between versions 1.1 and 2.0 of `mooLib`, it would look for the symbol in the library `;mooLib_1.1-2.0`. The advantage here is that the name of the library explicitly indicates the version of `mooLib` that introduced any particular export. A disadvantage is that since a new library name is added with each revision, the number of names may become unwieldy over time.

Use Reexport Libraries

A developer can split the functionality of a library into several libraries (for example, to reduce size or to isolate certain services). By using reexport libraries, older clients can use multiple newer libraries in place of an older one.

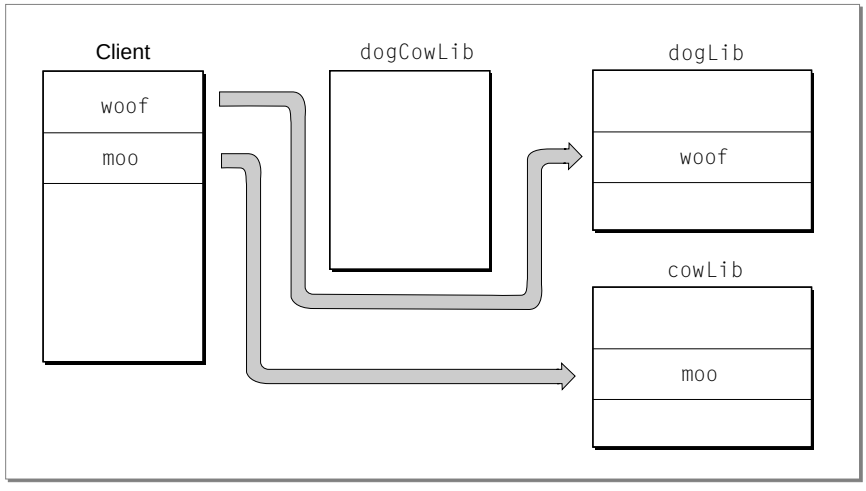
For example, say you have a library `dogCowLib` which has been split into two libraries `dogLib` and `cowLib`. Older clients still expect to import symbols from `dogCowLib`, so you must provide one. The new version of `dogCowLib` contains no code, however, but merely imports symbols from `dogLib` and `cowLib` and reexports them as its own. Figure 3-5 shows the use of a reexport library.

Figure 3-5 Using a reexport library



When the Code Fragment Manager performs relocations, `dogCowLib` is optimized out, with the result that symbol pointers point directly to `dogLib` or `cowLib`. Figure 3-6 shows the old client linked to the new libraries at runtime.

Figure 3-6 The reexport library removed at runtime



More generally, a developer can use reexport libraries to link against a collection of libraries that do not exist as real implementations. For example, you can group symbols in arbitrary libraries according to functionality during the build process and then use a reexport library at runtime to assign these symbols to the actual implementation libraries.

A drawback to using reexport libraries is that the client application receives all the connections associated with a reexport library even if they are not needed. In Figure 3-5, for example, even if the client application does not need any symbols in `dogLib`, the Code Fragment Manager prepares it anyway, since `dogCowLib` requires it.

Using the Main Symbol as a Data Structure

As mentioned before, the main symbol does not have to point to a routine, but can point to a block of data instead. You can use this fact to good effect with plug-ins, where the block of data referenced by the main symbol can contain essential information about the plug-in. Using the main symbol in this fashion has several advantages:

- The Code Fragment Manager returns the address of the main symbol when you programmatically prepare a fragment, so you do not need to call `FindSymbol`.
- You do not have to reserve and document the specific name of an export for your plug-in.

However, not having a specific symbol name means that the plug-in's purpose is not quite as obvious.

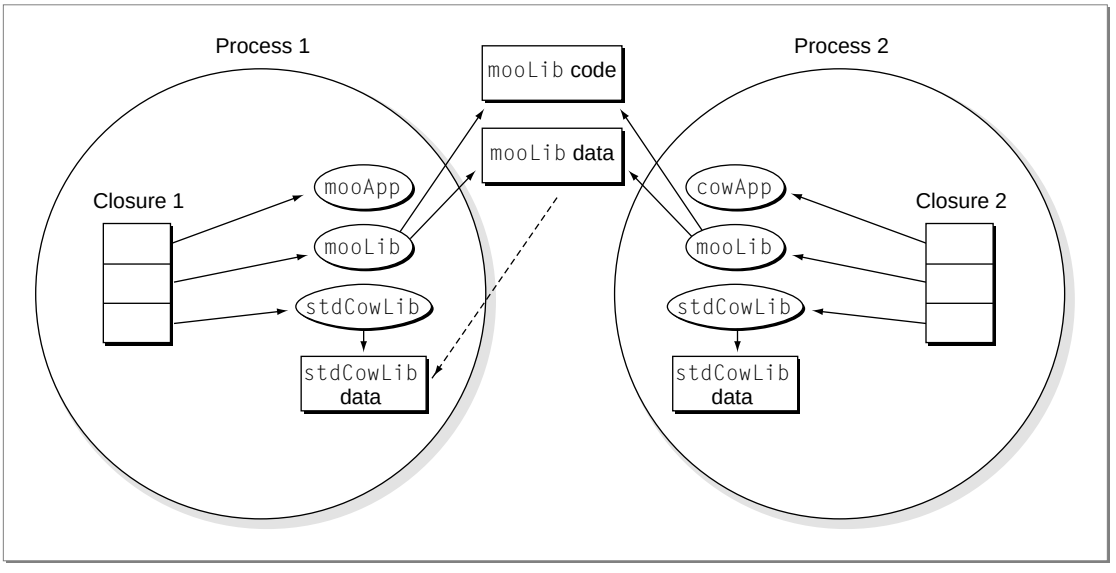
A plug-in can store its name, icon, or information about its symbols in the main symbol data structure. Storing symbolic information in this fashion eliminates the need for multiple `FindSymbol` calls.

Systemwide Sharing and Data-Only Fragments

As discussed in Chapter 1, a fragment can select either per-process or systemwide (global) sharing for its data sections. If you specify systemwide sharing, however, you should do so only with fragments that contain no code. The danger in having code in a fragment whose data is shared globally is that a globally shared routine may end up making a call into a process. Such a call

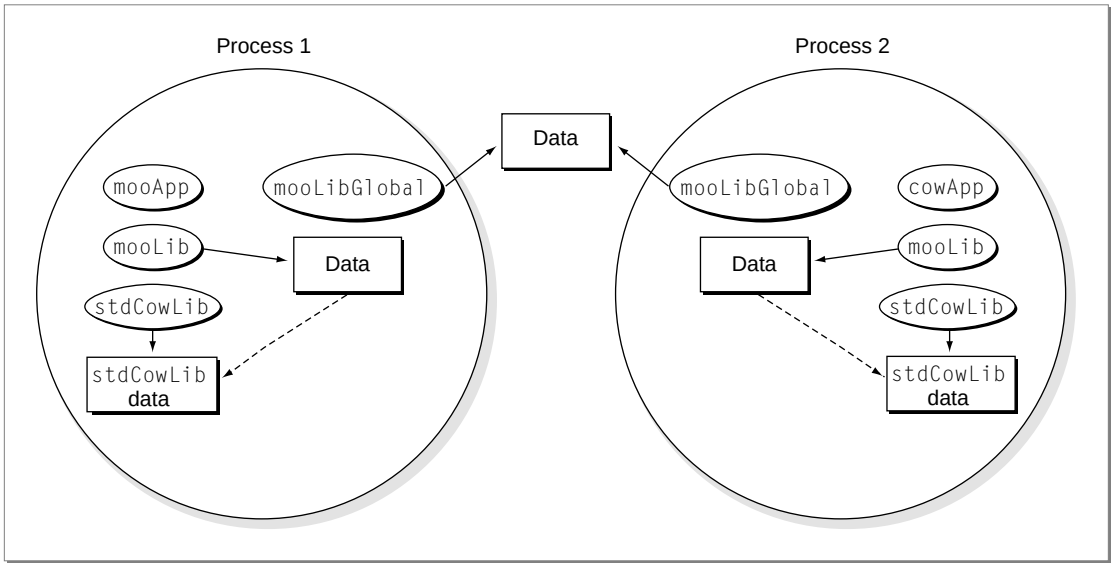
goes through the fragment's direct data area, which holds a pointer to the called routine's transition vector. If two or more processes are sharing the fragment, the target of the pointer can be unclear (each process could contain an eligible called routine). Figure 3-7 illustrates the problem.

Figure 3-7 Systemwide sharing in a fragment containing code and data



A solution is to isolate the data that must be globally shared in a data-only fragment. Function calls are stored in per-process data so there is no confusion as to which process the calls refer. Figure 3-8 shows the fragment `mooLib` separated into `mooLib` and `mooLibGlobal`.

Figure 3-8 Systemwide sharing using a data-only fragment

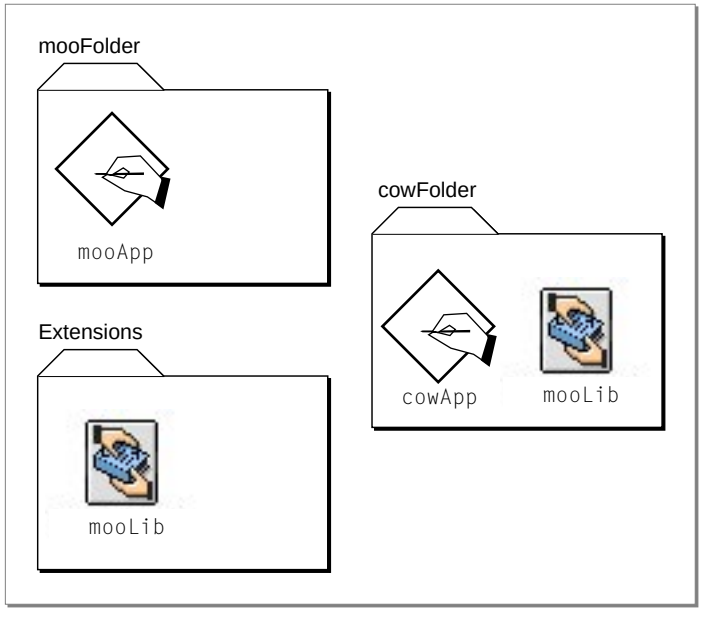


Multiple Fragments With the Same Name

The Code Fragment Manager associates fragments with physical entities (on disk, in memory, and so on) rather than names, even within the same closure. This referencing method means that it is possible to have the Code Fragment

Manager prepare two fragments with the same name (which may or may not be identical). For example, consider Figure 3-9, which shows a hard disk that contains two separate copies of the import library `mooLib`.

Figure 3-9 Identical but independent fragments



When the application `mooApp` launches, the Code Fragment Manager determines that `mooApp` requires the import library `mooLib` and, following its search path, eventually finds a copy in the default system libraries folder (the Extensions folder, for example). This copy of `mooLib` is then bound to `mooApp`.

Later, you decide to launch the application `cowApp`, which also depends on the import library `mooLib`. However, in searching for `mooLib`, the Code Fragment Manager finds a copy of the library in the folder containing `cowApp`. Since this location takes precedence over the Extensions folder, the Code Fragment Manager binds this copy of `mooLib` to `mooApp`.

The result is that two separate copies of `mooLib` exist at the same time. Even though they share the same name (and may in fact be completely identical), they do not share data or code; as far as the Code Fragment Manager is concerned, they are two separate fragments. This can lead to subtle problems when the libraries have specified systemwide sharing of data. For example, even if both copies of `mooLib` specified systemwide data sharing, they would not share global data with each other. On the other hand, allowing multiple copies of a library to exist can be useful for test or debugging purposes. For example, `cowApp` could use a test copy of `mooLib` without disturbing the copy used by `mooApp`.

CHAPTER 4

PowerPC Runtime Conventions

Contents

This chapter covers specific low-level details of the PowerPC runtime environment, including the following:

- data storage types
- stack structure
- routine calling conventions

These conventions may be useful for low-level programming (if you are writing in assembly language, for example) or for optimizing higher-level code.

Data Types

Table 4-1 lists the binary data types and their sizes in the PowerPC runtime environment.

Table 4-1 Data types in the PowerPC runtime environment

Type	Size (bytes)	Alignment (bytes)	Range	Notes
`UInt8`	1	1	0 to 255	
`SInt8`	1	1	–128 to 127	
`SInt16`	2	2	–32,768 to 32,767	
`UInt16`	2	2	0 to 65,535	
`SInt32`	4	4	-2^{-31} to $2^{31}-1$	
`UInt32`	4	4	0 to $2^{32}-1$	
Boolean	1	1		0 = false, nonzero = true
`float`	4	4	$\pm(2^{-149}$ to $2^{127})$	IEEE 754 standard
`double`	8	8	$\pm(2^{-1074}$ to $2^{1023})$	IEEE 754 standard
Pointer	4	4	0 to FFFFFFFF	

All numeric and pointer data types are stored in big-endian format (that is, high bytes first, then low bytes). Signed integers use two's-complement representation.

Data Alignment

The PowerPC runtime environment supports multiple data alignment modes. These alignments fall into two categories:

- the **natural alignment,** which is the alignment of a data type when allocated in memory or assigned a memory address
- the **embedding alignment,** which is the alignment of a data type within a composite data item

For example, the alignment of a `UInt16` variable may differ from that of a `UInt16` data item embedded in a data structure.

Note
Data items passed as parameters in a routine call have their own special alignment rules. See "Routine Calling Conventions," beginning on page 4-11, for more information. ◆

Table 4-1 (page 4-3) shows the natural alignment of each data type, which is simply the size of the data type. This alignment is fixed.

In data structures, you can specify an embedding alignment that varies depending on the alignment mode selected. Typically you can select the alignment mode using compiler options or pragmas. Table 4-2 shows the possible alignment modes.

Table 4-2 Embedded alignment modes

Data type	PowerPC	68K	Packed	Natural
`SInt8`	1	1	1	1
`UInt8`				
Boolean				
`SInt16`	2	2	1	2
`UInt16`				
`SInt32`	4	2	1	4
`UInt32`				
`float`	4	2	1	4
`double`	4 or 8	2	1	8
Pointer	4	2	1	4
Composite	4 or 8	2	1	16

In all but the 68K alignment mode, the embedding alignment of a composite (for example, a data structure or an array) is determined by the largest embedding alignment of its members. The total size of a composite is rounded up to be a multiple of its embedded alignment.

In 68K alignment mode, the embedded alignment of a composite is always 2 bytes. The total size of the composite is rounded up to a multiple of two.

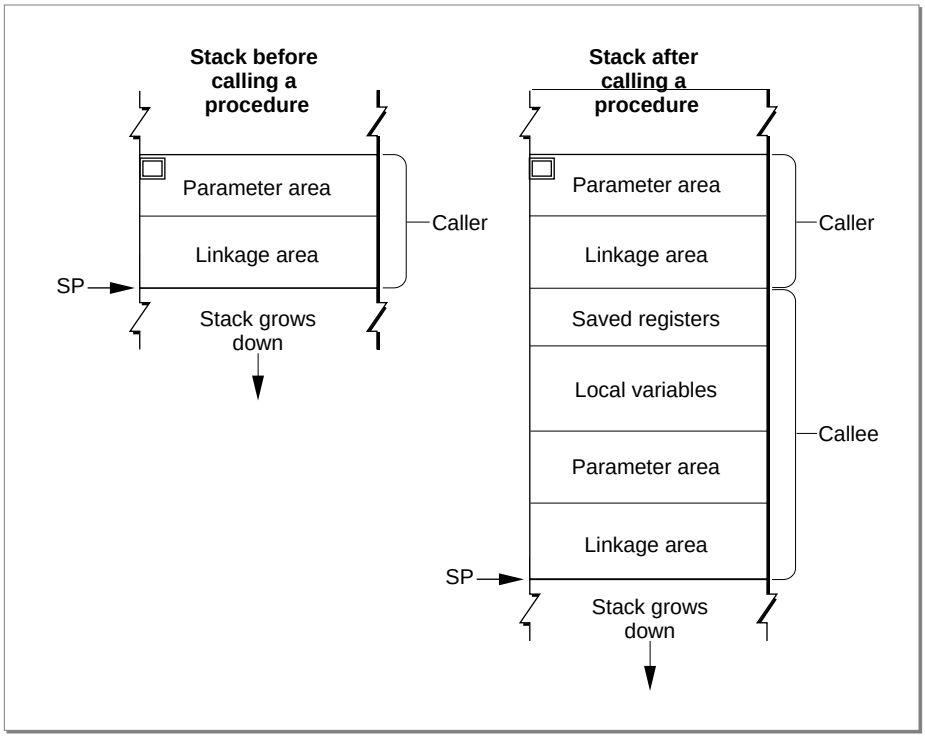
In PowerPC alignment mode, if the first embedded element in a data structure is type `double`, then the embedding alignment of all type `double` members in the structure is 8. In such cases, the embedding alignment for the entire structure is also 8 bytes.

Note that you may need to adjust embedded alignments if you are converting code from the classic 68K environment to the PowerPC (or CFM-68K) runtime environments. If you wish to enforce classic 68K alignment on your PowerPC code, you can often specify compiler pragmas or options to do so. Note, however, that the PowerPC processor is less efficient when accessing data that is not placed according to its natural alignment.

PowerPC Stack Structure

The PowerPC runtime environment uses a grow-down stack that contains linkage information, local variables, and a routine's parameter information as shown in Figure 4-1.

Figure 4-1 The PowerPC stack

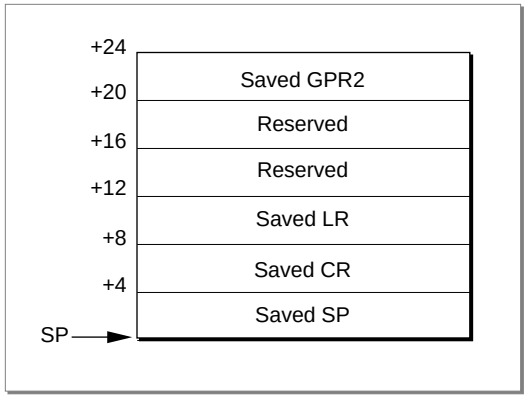


The typical PowerPC stack conventions use only a stack pointer (held in register GPR1) and no frame pointer. This configuration assumes a fixed stack frame size, which is known at compile time. Parameters are not passed by pushing them onto the stack.

The calling routine's stack frame includes a parameter area and some linkage information. The **parameter area** has space for the parameters of any routines the caller calls (*not* the parameters of the caller itself). Since the calling routine might call several different routines, the parameter area must be large enough to accomodate the largest parameter list of all the routines the caller calls. It is the calling routine's responsibility for setting up the parameter area before each call to some other routine, and the called routine's responsibility for accessing the parameters placed within it. See "Routine Calling Conventions," beginning on page 4-11, for more information about the calling conventions.

The calling routine's **linkage area** holds a number of values, some of which are saved by the calling routine and some by the called routine. Figure 4-2 shows the structure of the linkage area.

Figure 4-2 A stack frame's linkage area



The elements within the linkage area are as follows:

- The base register (GPR2) value is saved at `20(SP)` by the calling routine prior to the call if
  - the call is to an imported routine
  - the call is a pointer-based call (which may or may not be cross-fragment)

  This action ensures that the calling routine can still access its own direct data area upon return. See "PowerPC Implementation," beginning on page 2-8, for more information. Local calls do not need to save this value.

- The Link Register (LR) value is saved at `8(SP)` by the *called routine* if it chooses to do so.
- The Condition Register (CR) value may be saved at `4(SP)` by the *called routine*. As with the Link Register value, the called routine is not required to save this value.
- The stack pointer is always saved by the calling routine as part of its stack frame.

Note that the linkage area is at the top of the stack, adjacent to the stack pointer. This positioning is necessary so the calling routine can find and restore the values stored there and also to enable the called routine to find the caller's parameter area. This placement means that a routine cannot push and pop parameters from the stack once the stack frame is set up.

The stack frame also includes space for the called routine's local variables. In general, the general-purpose registers GPR13 through GPR31 and the floating-point registers FPR14 through FPR31 are reserved for the routine's local variables. However, if the routine contains more local variables than would fit in the registers, it uses additional space on the stack. The size of the local variable area is determined at compile time; once a stack frame is allocated, the size of the local variable area cannot change.

Prologs and Epilogs

The called routine is responsible for allocating its own stack frame, making sure to preserve 16-byte alignment on the stack. This action is accomplished by the **prolog** before entering the actual routine. The compiler-generated prolog code does the following:

- Decrements the stack pointer to account for the new stack frame.
- Writes the previous value of the stack pointer to its own linkage area. This procedure ensures that the stack can be restored to its original state after returning from the call.
- Saves all nonvolatile general-purpose and floating-point registers into the saved-registers area. Note that if the called routine does not change a particular nonvolatile register, it does not save it.
- Saves the Link Register and Condition Register values in the caller's linkage area, if needed.

Note
The order in which the prolog executes these actions is determined by convention, not by any requirements of the PowerPC runtime architecture. ◆

Listing 4-1 shows some sample prolog code. Note that the order of these actions differs from the order previously described.

Listing 4-1 Sample prolog code

```
linkageArea: set 24             ; size in PowerPC environment
params: set 32                  ; callee parameter area
localVars: set 0                ; callee local variables
numGPRs: set 0                  ; volatile GPRs used by callee
numFPRs: set 0                  ; volatile FPRs used by callee)

spaceToSave: set linkageArea + params + localVars
spaceToSave: set spaceToSave + 4*numGPRs + 8*numFPRs

.moo:                           ; PROLOG
    mflr    r0,                 ; extract return address
    stw     r0,8(SP)            ; save the return address
    stwu    SP, -spaceToSave(SP); skip over caller save area
```

After the called routine exits, the **epilog** code executes, which does the following:

- Restores the nonvolatile general-purpose and floating-point registers that were saved in the stack frame.
- Restores the Condition Register and Link Register values that were stored in the linkage area.
- Restores the stack pointer to its previous value.
- Returns to the calling routine using the address stored in the Link Register.

Listing 4-2 shows some sample epilog code.

Listing 4-2 Sample epilog code

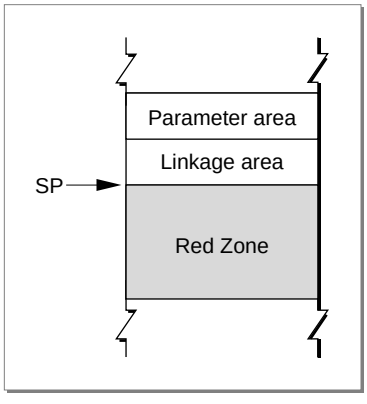
```
                                    ; EPILOG
lwz     r0,spaceToSave(SP)+8        ; get the return address
mtlr    R0                          ; reset Link Register
addic   SP,SP,spaceToSave           ; restore stack pointer
blr                                 ; return
```

The calling routine is responsible for restoring its GPR2 value immediately after returning from the called routine.

The Red Zone

The space beneath the stack pointer, where a new stack frame would normally be allocated, is called the **Red Zone.** This area, as shown in Figure 4-3, may be used for any purpose as long as a new stack frame does not need to be added to the stack.

Figure 4-3 The Red Zone



For example, the Red Zone may be used by a **leaf procedure.** A leaf procedure is a routine that does not call any other routines. Since it does not call any other routines, it does not need to allocate a parameter area on the stack. Furthermore, if it does not need to use the stack to store local variables, it need save and restore only the nonvolatile registers that it uses for local variables. Since by

definition no more than one leaf procedure is active at any time, there is no possibility of multiple leaf procedures competing for the same Red Zone space.

A leaf procedure does not allocate a stack frame nor does it decrement the stack pointer. Instead it stores the Link Register and Condition Register values in the linkage area of the routine that calls it (if necessary) and stores the values of any nonvolatile registers it uses in the Red Zone. This streamlining means that a leaf procedure's prolog and epilog do only minimal work; they do not have to set up and take down a stack frame.

When an exception handler is called, the Exception Manager automatically decrements the stack pointer by 224 bytes (the largest possible area used to save registers), to skip over any possible Red Zone information, and then restores the stack pointer when the handler exits. The Exception Manager does this because an exception handler cannot know in advance if a leaf procedure is executing at the time the exception occurs. If you are writing code that modifies the stack at interrupt time, you must similarly decrement the stack pointer by 224 bytes to preserve any Red Zone information and restore it after the interrupt call.

Note
The value of 224 bytes is the space occupied by nineteen 32-bit general-purpose registers plus eighteen 64-bit floating-point registers, rounded up to the nearest 16-byte boundary. If a leaf procedure's Red Zone usage would exceed 224 bytes, then it must set up a stack frame just like routines that call other routines. ◆

Routine Calling Conventions

This section details the process of passing parameters to a routine in the PowerPC runtime environment.

Note
These parameter passing conventions are part of Apple's standard for procedural interfaces. Object-oriented languages may use different rules for their own method calls. For example, the conventions for C++ virtual function calls may be different from those for C functions. ◆

A routine can have a fixed or variable number of arguments. In an ANSI-style C syntax definition, a routine with a variable number of arguments typically appears with ellipsis points (…) at the end of its input parameter list.

A variable-argument routine may have several required (that is, fixed) parameters preceding the variable parameter portion. For example, the routine definition

```
mooColor(number, [color1...])
```

gives no restriction on the number of *color* arguments, but you must always precede them with a *number* argument. Therefore, *number* is a fixed parameter.

Typically the calling routine passes parameters in registers. However, the compiler generates a parameter area in the caller's stack frame that is large enough to hold all parameters passed to the called routine, regardless of how many of the parameters are actually passed in registers. There are several reasons for this scheme:

- It provides the callee with space to store a register-based parameter if it wants to use one of the parameter registers for some other purpose (for instance, to pass parameters to a subroutine).
- Routines with variable-length parameter lists must often access their parameters from RAM, not from registers. Such routines must reserve eight registers (32 bytes) in the parameter area to hold the parameter values.
- To simplify debugging, some compilers may write parameters from the parameter registers into the parameter area in the stack frame; this allows you to see all the parameters by looking only at that parameter area.

You can think of the parameter area as a data structure that has space to hold all the parameters in a given call. The parameters are placed in the structure from left to right according to the following rules:

- All parameters are aligned on 4-byte (word) boundaries.
- Noncomposite parameters smaller than 4 bytes occupy the low order bytes of their word.
- Composite parameters (such as data structures) are followed by padding to make a multiple of 4 bytes, with the padding bytes being undefined.

For a routine with fixed parameters, the first 8 words (32 bytes) of the data structure, no matter the size of the individual parameters, are passed in registers according to the following rules:

- The first 8 words are placed in GPR3 through GPR10 unless a floating-point parameter is encountered.
- Floating-point parameters are placed in the floating-point registers FPR1 through FPR13.
- If a floating-point parameter appears before all the general-purpose registers are filled, the corresponding GPRs that match the size of the floating-point parameter are skipped. For example, a `float` item causes one (4-byte) GPR to be skipped, while an item of type `double` causes two GPRs to be skipped.
- If the number of parameters exceeds the number of usable registers, the calling routine writes the excess parameters into the parameter area of its stack frame.

Note
Currently the parameter area must be at least 8 words (32 bytes) in size. ◆

For example, consider a routine `mooFunc` with this declaration:

```
void mooFunc (SInt32 i1, float f1, double d1, SInt16 s1, double d2,
              UInt8 c1, UInt16 s2, float f2, SInt32 i2);
```

To see how the parameters of `mooFunc` are arranged in the parameter area on the stack, first convert the parameter list into a structure, as follows:

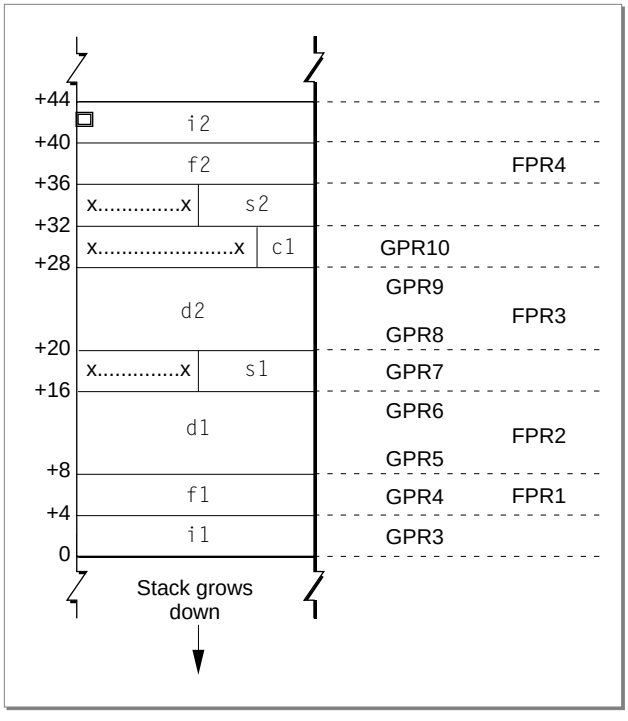
```
struct params {
    SInt32                  p_i1;
    float                   p_f1;
    double                  p_d1;
    SInt16                  p_s1;
    double                  p_d2;
    UInt8                   p_c1;
    UInt16                  p_s2;
    float                   p_f2;
    SInt32                  p_i2;
};
```

This structure serves as a template for constructing the parameter area on the stack. (Remember that, in actual practice, many of these variables are passed in

registers; nonetheless, the compiler still allocates space for all of them on the stack, for the reasons just mentioned.)

The "top" position on the stack is for the field `pi_1` (the structure field corresponding to parameter `i1`). The floating-point field `p_f1` is assigned to the next word in the parameter area. The 64-bit double field `p_d1` is assigned to the next two words in the parameter area. Next, the short integer field `p_s1` is placed into the following 32-bit word; the original value of `p_s1` is in the lower half of the word, and the padding is in the upper half. The remaining fields of the `params` structure are assigned space on the stack in exactly the same way, with unsigned values being extended to fill each field to make it a 32-bit word. The final arrangement of the stack is illustrated in Figure 4-4. (Because the stack grows down, it looks as though the fields of the `params` structure are upside down.)

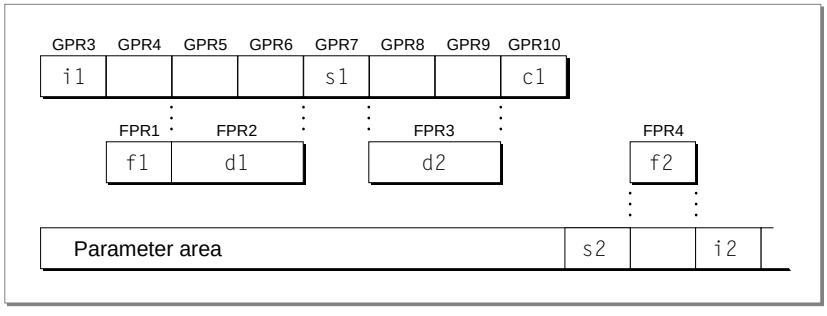
Figure 4-4 The organization of the parameter area of the stack



To see which parameters are passed in registers and which are passed on the stack, you need to map the stack, as illustrated in Figure 4-4, to the available general-purpose and floating-point registers. Therefore, the parameter `i1` is passed in GPR3, the first available general-purpose register. The floating-point parameter `f1` is passed in FPR1, the first available floating-point register. This action causes GPR4 to be skipped.

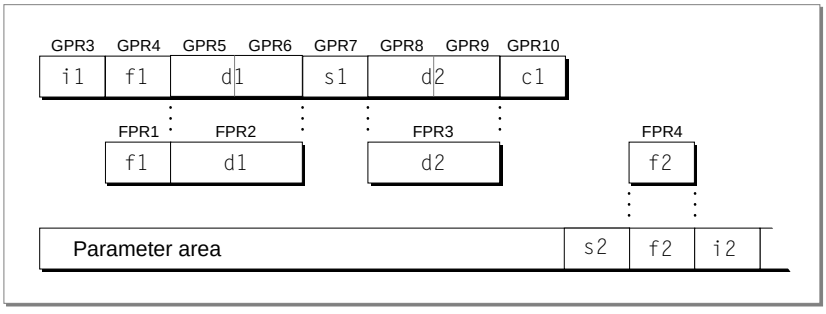
The parameter `d1` is placed into FPR2 and the corresponding general-purpose registers GPR5 and GPR6 are unused. The parameter `s1` is placed into the next available general-purpose register, GPR7. Parameter `d2` is placed into FPR3, with GPR8 and GPR9 masked out. Parameter `c1` is placed into GPR10, which fills out the first 8 words of the data structure. Parameter `s2` is then passed in the parameter area of the stack. Parameter `f2` is passed in FPR4, since there are still floating-point registers available. Finally, parameter `i2` is passed on the stack. Figure 4-5 shows the final layout of the parameters in the registers and the parameter area.

Figure 4-5 Parameter layout in registers and the parameter area



If you have a C routine with a variable number of parameters (that is, one that does not have a fixed prototype), the compiler cannot know whether to pass a parameter in the variable portion of the routine in the general-purpose (that is, fixed-point) registers or in the floating-point registers. Therefore, the compiler passes the parameter in both the floating-point and the general-purpose registers, as shown in Figure 4-6.

Figure 4-6 Passing a variable number of parameters



The called routine can access parameters in the fixed portion of the routine definition as usual. However, in the variable-argument portion of the routine, the called routine must copy the GPRs to the parameter area and access the values from there. Listing 4-3 shows a routine that accesses values by walking through the stack.

Listing 4-3 A variable-argument routine

```
double dsum (int count, ...)
{
    double sum = 0.0;
    double * arg = (double *) (&count + 1 /* pointer arithmetic */);
    while (count > 0 ) {
        sum += *arg;
        arg += 1;            /* pointer arithmetic */
        count -= 1;
    }
return sum;
}
```

Function Return

In the PowerPC runtime environment, floating-point function values are returned in register FPR1 (or FPR1 and FPR2 for long double values). Other values are returned in GPR3 as follows:

- Functions returning simple values smaller than 4 bytes (such as type `SInt8`, `Boolean`, or `SInt16`) place the return value in the least significant byte or bytes of GPR3. The most significant bytes in GPR3 are undefined.
- Functions returning 4-byte values (such as pointers, including array pointers, or types `SInt32` and `UInt32`) return them normally in GPR3.
- If a function returns a composite value (for example, a `struct` or `union` data type) or a value larger than 4 bytes, a pointer must be passed as an implicit left-most parameter before passing all the user-visible arguments (that is, the address is passed in GPR3, and the actual parameters begin with GPR4). The address of the pointer must be a memory location large enough to hold the function return value. Since GPR3 is treated as a parameter in this case, its value is not guaranteed on return.

Register Preservation

Table 4-3 lists registers used in the PowerPC runtime environment and their volatility in routine calls. Registers that retain their value after a routine call are called *nonvolatile*. All registers are 4 bytes long.

Table 4-3 Volatile and nonvolatile registers

Type	Register	Preserved by a routine call?	Notes
General-purpose register	GPR0	No	
	GPR1	See next column	Used as the stack pointer to store parameters and other temporary data items.

continued

Table 4-3 Volatile and nonvolatile registers (continued)

Type	Register	Preserved by a routine call?	Notes
General-purpose register *(continued)*	GPR2	See next column	Used as the base register to access the direct data area. GPR2 is preserved by direct calls; for indirect calls the caller must restore the value after the call.
	GPR3	See next column	Holds the return value or the address of the return value in function calls. For routine calls that do not return a value, GPR3 is used to pass parameter values.
	GPR4-GPR10	No	Used to pass parameter values in routine calls.
	GPR11-GPR12	No	
	GPR13-GPR31	Yes	
Floating-point register	FPR0	No	
	FPR1-FPR13	No	Used to pass floating-point parameters in routine calls.
	FPR14-FPR31	Yes	
Link Register	LR	No	Stores the return address of the calling routine during a routine call.
Count Register	CTR	No	

continued

Table 4-3 Volatile and nonvolatile registers (continued)

Type	Register	Preserved by a routine call?	Notes
Fixed-point exception register	XER	No	
Condition Registers	CR0-CR1	No	
	CR2-CR4	Yes	
	CR5-CR7	No	

CFM-68K Runtime Conventions

Contents

This chapter covers data storage and parameter-passing conventions for the CFM-68K runtime environment. All CFM-68K runtime conventions are language independent. These conventions may be useful for low-level programming (if you are writing in assembly language, for example) or for optimizing higher-level code.

Data Types

Table 5-1 lists the binary data types and their sizes in the CFM-68K runtime environment. These types and sizes are identical to those in the PowerPC runtime environment.

Table 5-1 Data types in the CFM-68K runtime environment

Type	Size (bytes)	Alignment (bytes)	Range	Notes
`UInt8`	1	1	0 to 255	
`SInt8`	1	1	–128 to 127	
`SInt16`	2	2	–32,768 to 32,767	
`UInt16`	2	2	0 to 65,535	
`SInt32`	4	4	-2^{-31} to $2^{31}-1$	
`UInt32`	4	4	0 to $2^{32}-1$	
Boolean	1	1		0 = false, nonzero = true
`float`	4	4	$\pm(2^{-149}$ to $2^{127})$	IEEE 754 standard
`double`	8	8	$\pm(2^{-1074}$ to $2^{1023})$	IEEE 754 standard
Pointer	4	4	0 to FFFFFFFF	
`extended`	10 or 12	4		SANE or MC68881 data type

All numeric and pointer data types are stored in big-endian format (that is, high bytes first, then low bytes). Signed integers use two's-complement representation.

IMPORTANT
The layout of the `extended` data type is either that of the SANE 80-bit data type or that of the 96-bit MC68881 data type, depending on the software development environment used. Because of this variability, you should not use the `extended` data type for imported or exported routines or data. ▲

The size of data structures and unions must be a multiple of two, and an extra byte may be added at the end to meet this requirement. Items inside a data structure (except for types `UInt8` and `SInt8`) are placed on a 2-byte boundary with an extra padding byte inserted if necessary. Type `UInt8` and type `SInt8` items (single variables or arrays) are merely placed in the next available byte.

Routine Calling Conventions

This section details the process of passing parameters to a routine in the CFM-68K runtime environment.

Note
These parameter passing conventions are part of Apple's standard for procedural interfaces. Object-oriented languages may use different rules for their own method calls. For example, the conventions for C++ virtual function calls may be different from those for C functions. ◆

A routine can have a fixed or variable number of arguments. In an ANSI-style C syntax definition, a routine with a variable number of arguments typically appears with ellipsis points (…) at the end of its input parameter list.

A variable-argument routine may have several required (that is, fixed) parameters preceding the variable parameter portion. For example, the function definition

`mooColor(`*number*`,[`*color1*`...])`

gives no restriction on the number of *color* arguments, but you must always precede them with a *number* argument. Therefore, *number* is a fixed parameter.

The calling routine passes parameters by pushing their values onto the stack, and the stack grows downward (towards lower addresses) with each push. The rightmost parameter is the first pushed onto the stack, with the others following from right to left. For example, given the code

```
cow = mooFunc(moo1, moo2, moo3);
```

the calling routine first pushes the value of `moo3` onto the stack, followed by `moo2` and then `moo1`.

The return address of the routine is the last item pushed onto the stack.

Note
The order of passing parameters onto the stack in CFM-68K is identical to that for the classic 68K C calling convention. For information about the 68K stack structure, see "Classic 68K Stack Structure and Calling Conventions," beginning on page 11-4. ◆

Parameter Deallocation

In the CFM-68K runtime environment, responsibility for removing items from the stack depends on the function type.

- In a fixed-argument type routine, the called routine deallocates (that is, pops from the stack) the return address and all the passed parameters before, or as part of, its return. The calling routine does not need to do any cleanup.
- If the called routine is a variable-argument type, it only pops the return address before returning. The calling routine must then deallocate all the parameters it pushed onto the stack.

Stack Alignment

To improve performance, the CFM-68K runtime architecture requires a 4-byte (minimum) alignment for all parameters pushed onto the stack. This applies to stack space used in function prologs (that is, stack space reserved for automatic memory variables and temporaries) as well as space allocated using the `alloca` dynamic stack allocation operation. Types `UInt8`, `SInt8`, `Boolean`, `UInt16`, and

SInt16 parameters are passed in the least significant byte or bytes with padding added. Data types struct, union, and extended are passed in the most significant bytes, with padding added afterwards if necessary.

Fixed-Argument Passing Conventions

Fixed parameters may either be items used to call a fixed-argument type routine, or fixed items that precede the variable items in a variable-argument function call. In either case, fixed parameters must occupy a multiple of 4 bytes when pushed onto the stack, with padding added if necessary. Note that the data can actually be pushed in any order as long as the final alignment matches the required convention.

- Parameters of type UInt8, SInt8, and Boolean are pushed onto the stack as 1 byte of data (the least significant byte) along with 3 bytes of undefined padding.
- Parameters of type UInt16 and SInt16 are pushed onto the stack as 2 bytes (least significant) of data plus 2 bytes of padding.
- Pointers to procedures and arrays are pushed normally (since they are 4 bytes long), as are UInt32, SInt32, and float data items.
- Type double parameters are passed by pushing the memory image of the 8-byte item onto the stack.
- Type extended parameters can be either 10 or 12 bytes long, depending on the development environment. For 10-byte items, 2 padding bytes are pushed onto the stack before pushing the parameter. A 12-byte extended data item is pushed onto the stack normally (since it is a multiple of 4 bytes).
- If the size of a data structure or union is not a multiple of 4 bytes, 2 padding bytes are added to the stack before pushing the parameter. Otherwise, the parameter is pushed onto the stack normally. In both cases, the memory image of the item is passed.
- Bit field layout is not defined. You should not use bit fields in procedures or data structures that have shared library interfaces.

Variable-Argument Passing Conventions

When passing variable arguments, padding is added to some data types when pushing them onto the stack:

- Parameters of types `UInt8`, `SInt8`, `UInt16`, `SInt16`, and `Boolean` are converted to type `SInt32` (as if by assignment) and pushed onto the stack as a 4-byte integer data item.
- Both `float` and `double` parameters are pushed onto the stack as 8-byte `double` data items (`float` data types are converted to type `double` (as if by assignment) before being pushed).
- All other data types are passed normally.

Function Value Return

In the CFM-68K runtime environment, the placement of the return value depends on its size:

- Functions returning `UInt8`, `SInt8`, or `Boolean` data types place the return value in the least significant byte of D0. The three most significant bytes in D0 are undefined.
- Functions returning `UInt16` or `SInt16` data types place the return value in the two least significant bytes of D0. The two most significant bytes in D0 are undefined.
- Functions returning pointers (including array pointers), `UInt32`, `SInt32`, or `float` data types place the return value in D0.
- Functions returning small data structures or `union` data types place them in the least significant bytes of D0. For example, a 4-byte structure takes up D0, while a 2-byte structure occupies the two least significant bytes of D0, with the extra bytes being undefined.
- If the function return value is larger than 4 bytes (this applies to `double` and `extended` data types, as well as to large `struct` or `union` data types), a pointer must be pushed onto the stack at call time after all the user-visible arguments have been pushed. The address of the pointer must be a memory location large enough to hold the function return value. When the function exits, it returns this address in the D0 register.

Stack Frames, A6, and Reserved Frame Slots

The CFM-68K runtime architecture requires two long words in the stack frame to be reserved (that is, unused) for future use. The word locations for these reserved slots are `-4(A6)` and `-8(A6)`.

Routines making calls through procedure pointers (that is, indirect or cross-fragment calls) must have an A6 frame and must reserve the two long words. Leaf routines or routines that make only direct (in-fragment) calls do not need to use A6 as a link, and they do not require reserved stack frame slots. However, some debugging options may require you to set up a stack frame.

In general, you should not use the A6 register except as a frame pointer, and if you do set up an A6 stack frame, you must also reserve the two long frame slots.

Register Preservation

Table 5-2 lists registers used in the CFM-68K runtime environment and their volatility in function calls. Registers that retain their value after a routine call are called *nonvolatile*. All registers are 4 bytes long.

Table 5-2 Volatile and nonvolatile registers

Type	Register	Preserved by a routiine call?	Notes
Data register	D0 through D2	No	
	D3 through D7	Yes	
Address register	A0	No	
	A1	No	Used to pass transition vector addresses (+4) when making indirect or cross-fragment calls.
	A2 through A4	Yes	

continued

Table 5-2 Volatile and nonvolatile registers (continued)

Type	Register	Preserved by a routiine call?	Notes
Address register *(continued)*	A5	See next column	Used to access global data objects and the jump table. A5 is preserved by direct (in-fragment) calls, but not by cross-fragment or indirect calls.
	A6	Yes	Used as the back link and frame pointer when making cross-fragment calls.
	A7	See next column	A7 is the stack pointer used to push and pop parameters and other temporary data items
Floating-point register	F0 through F3	No	When present.
	F4 through F7	Yes	When present.
Condition Register	CR	No	Bits are set by compare instructions and used for conditional branching.

The Mixed Mode Manager

Contents

In certain cases, your CFM-based application or shared library may need to call routines written in classic 68K code or vice versa. For example, a PowerPC runtime program may need to call a system software routine that runs as emulated classic 68K code. The Mixed Mode Manager allows you to make such routine calls transparently.

You should read this chapter if you have any of the following concerns:

- You are writing CFM-based code, but want to make sure that it remains compatible with existing classic 68K software (third-party plug-ins, for example).
- You need to maintain binary compatibility with old classic 68K routines or libraries whose source code is not available. You cannot recompile them for the CFM-based architecture, but you still want to be able to use the old routines.
- You are writing a low-level debugger or other tool that requires understanding of the Mixed Mode Manager.

This chapter assumes you have general programming knowledge of both the CFM-based runtime architecture and the classic 68K runtime architecture.

Overview

The Mixed Mode Manager is essentially a "black box" interface that allows routines with different calling conventions to exchange parameter information. The routines may reflect different architectures, different hardware, or both, but the basic treatment is the same.

In addition to routine parameter information, the Mixed Mode Manager requires the following information to make a mode switch:

- the calling conventions of the routine making the call
- a translation key that tells the Mixed Mode Manager how to manipulate the parameters to meet the calling conventions of the routine being called

Currently the Mixed Mode Manager handles calls between CFM-based architecture code and classic 68K architecture code.

The Mixed Mode Manager has two rules for the code that it handles:

1. The older classic 68K code does not need to be aware of the Mixed Mode Manager and no modification is required for mode switching.
2. The newer CFM-based code *must* be aware of the Mixed Mode Manager and take the steps necessary to invoke it if there is the possibility of a mode switch. Typically a mode switch can occur when a routine calls code not stored directly in the application or software (for example when loading and executing code stored in a resource).

Given these assumptions, if there is the possibility of a mode switch, there are four different types of calls that can be made:

- A CFM-based routine calls a classic 68K routine.
- A CFM-based routine calls a CFM-based routine.
- A classic 68K routine calls a CFM-based routine.
- A classic 68K routine calls a classic 68K routine.

How the call is handled depends on the type of code that originates the call:

1. A CFM-based routine originates the call.

Rule 2 requires the calling CFM-based routine to invoke the Mixed Mode Manager if there is a possibility of a mode switch.

If the Mixed Mode Manager discovers that the called routine is classic 68K, a mode switch is required. The Mixed Mode Manager should make the switch and then execute the call. Any return values are passed back to the calling routine.

If the Mixed Mode Manager discovers that the called routine is CFM-based, no mode switch is necessary. The Mixed Mode Manager should allow the call to be made normally.

Note that you need to call the Mixed Mode Manager only if the calling conventions of the called routine are unknown. If you know that both the calling routine and the called routine are CFM-based (when making routine calls within an application, for example), you do not have to call the Mixed Mode Manager.

2. A classic 68K routine originates the call.

Rule 1 requires that a classic 68K routine need no knowledge of the Mixed Mode Manager.

If the called routine is CFM-based, the supplier of the CFM-based routine must make sure the Mixed Mode Manager is invoked. The Mixed Mode Manager can then make the mode switch, execute the call, and pass back any return values.

If the called routine is a classic 68K routine, neither the calling routine or the called routine invokes the Mixed Mode Manager. The call proceeds normally.

IMPORTANT

The Mixed Mode Manager knows the size of the parameters it translates but not their type, so it cannot handle floating point parameters. If you need to pass floating-point values in a possible Mixed Mode call, you should pass pointers to the values instead. ▲

Universal Procedure Pointers and Routine Descriptors

While the Mixed Mode Manager is the mechanism for switching between CFM-based code and classic 68K code, the actual interface between the two types of code is the **universal procedure pointer.** A universal procedure pointer may be either of the following:

- A pointer to classic 68K code.
- A pointer to a **routine descriptor,** a data structure that describes the address of the called routine, its parameter signature, and its calling conventions. The Mixed Mode Manager uses the routine descriptor as a key to translate between the CFM-based and classic 68K calling conventions.

Both the calling code and the supplier of the called routine must agree to pass universal procedure pointers to each other. In general, you do not have to worry about which flavor of universal procedure pointer you are passing; as long as you pass a pointer of type `UniversalProcPtr`, the Mixed Mode Manager handles the rest and makes the mode switch when necessary. How you set up a universal procedure pointer varies depending on the type of code that initiates the call.

Note
Note that the choices for universal procedure pointers reflect the Mixed Mode Manager rules described earlier. Classic 68K code can pass universal procedure pointers without any code modification because all classic 68K procedure pointers are simply redefined to be universal procedure pointers. ◆

CFM-Based Code Originates the Call

If CFM-based code makes a pointer-based call to a routine that might be in classic 68K code, you should call the routine `CallUniversalProc` to invoke a universal procedure pointer instead of a standard procedure pointer. For example, instead of simply calling an external routine using

```
(*moo)(cow);
```

you must call

```
CallUniversalProc((UniversalProcPtr) moo, mooProcInfo, cow);
```

where `mooProcInfo` describes the calling conventions of `moo`. See *Inside Macintosh: PowerPC System Software* for more information on setting up the `ProcInfo` data structure.

IMPORTANT
In general you need to call `CallUniversalProc` only when calling external routines. Most CFM-based to CFM-based calls (including pointer-based calls) know that the called routine is CFM-based, so they do not need to call `CallUniversalProc`. ▲

Calling `CallUniversalProc` invokes the Mixed Mode Manager, which decides if a mode switch is necessary.

If the pointer it received (`*moo` in this case) turns out to point to a routine descriptor, the call requires a mode switch.The Mixed Mode Manager uses the routine descriptor to translate the parameter information into the form that the classic 68K routine expects to see and then calls the routine. After executing the routine, any return values are translated and passed back to the calling CFM-based routine.

If no mode switch is necessary the Mixed Mode Manager allows the call to be made normally. When the called routine returns, control passes back directly to the caller, not the Mixed Mode Manager.

Classic 68K Code Originates the Call

If classic 68K code initiates the call, then the calling routine is not required to take any action; it is never even aware that a mode switch might be necessary.

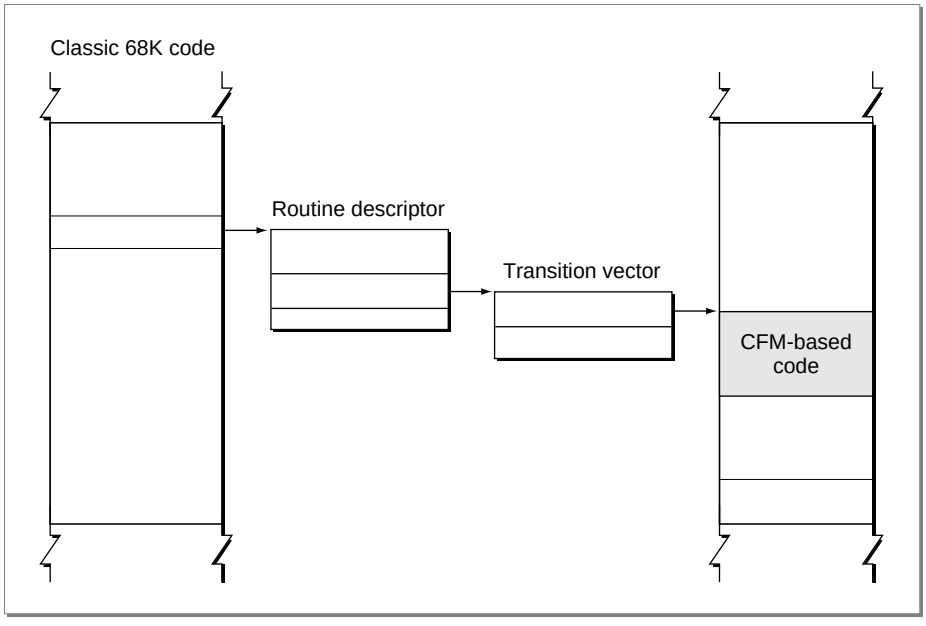
The calling routine must always pass a universal procedure pointer. If the called routine is also classic 68K code, the universal procedure pointer is simply a classic 68K pointer (that is, a pointer to the called routine). The Mixed Mode Manager is never invoked, and the call proceeds normally.

If the called routine is CFM-based code, the universal procedure pointer cannot be a classic 68K pointer; it must therefore be a pointer to a routine descriptor. The classic 68K caller is not required to change, so the supplier of the CFM-based routine must provide the routine descriptor.

Note
The routine descriptor is not part of the called routine. Rather, it is a shell or wrapper through which all external calls to the routine must pass. ◆

The first instruction in the routine descriptor is an A-line instruction that invokes the Mixed Mode Manager. The Mixed Mode Manager handles the mode switch using the information stored in the routine descriptor and then calls the transition vector of the CFM-based code. Figure 6-1 shows the calling path from the classic 68K code to the CFM-based code.

Figure 6-1 Calling path from classic 68K code to a CFM-based routine



After the call any return values are passed back to the classic 68K caller.

In order to satisfy the agreement to always pass universal procedure pointers, you must create routine descriptors for any CFM-based routines that may be called by classic 68K code. For example, if you supply a callback routine, you must take additional steps to anticipate a possible mode switch when the callback occurs. A classic 68K runtime function call such as

```
AEInstallEventHandler (kCoreEventClass, kAEOpenApplication,
                           HandleOapp,0,false);
```

must be changed to

```
UniversalProcPtr myHandleOappProc;
myHandleOappProc = NewAEEventHandlerProc (HandleOapp);
AEInstallEventHandler (kCoreEventClass,kAEOpenApplication,
                           myHandleOappProc,0,false)
```

The `NewAEEventHandlerProc` macro (defined in `AppleEvents.h`) calls the Mixed Mode Manager's `NewRoutineDescriptor` function to create a routine descriptor for `HandleOapp`.

Note
In certain cases where you cannot modify the CFM-based code (if it is a third-party library whose source code is unavailable, for example), it is possible to construct routine descriptors in your classic 68K code. ◆

Mixed Mode Manager Performance Issues

The Mixed Mode Manager, while extremely useful for maintaining compatibility between CFM-based code and classic 68K code, takes a significant number of instruction cycles to perform a mode switch, so you should keep this in mind when determining when and how often to switch between architectures.

In general this is not a problem if the time spent switching architectures is a negligible percentage of the time spent in the called routines. For example, if your classic 68K application calls a PowerPC graphics filter plug-in, most of the execution time is spent crunching numbers in the plug-in, so performance is not affected.

However, consider a short PowerPC patch for an emulated classic 68K software program. Theoretically increasing the amount of native PowerPC code should improve performance. However, if the mode-switching time is a significant portion of the patch's execution time and the patch is in a location where it is called frequently, considerable "dead time" accumulates as the Mixed Mode Manager switches back and forth; in such cases performance can actually decrease. In extreme cases, the time spent mode switching is so great that a classic 68K version of the patch results in better performance than a PowerPC patch. To avoid such problems, you should create fat patches that contain both CFM-based and classic 68K code. See "Mode Switching Implementations," beginning on page 6-10, and "Accelerated and Fat Resources," beginning on page 7-4, for more information on creating fat programs.

Mode Switching Implementations

This section describes the implementations that the Mixed Mode Manager uses to switch modes between PowerPC and emulated classic 68K and in switching between CFM-68K and classic 68K.

Note that you need to read this section only if you need low-level details of how the Mixed Mode Manager implements stack switch frames during a mode switch (if you are writing a debugger, for example).

Calling PowerPC Code From Classic 68K Code

This section describes how the Mixed Mode Manager switches modes from the classic 68K emulated environment to the PowerPC native environment. This can happen when classic 68K code calls a system software routine or plug-in that is implemented in the PowerPC instruction set.

Suppose that a classic 68K application calls a PowerPC routine. The application is not aware that it is running under the 68LC040 Emulator, so it just pushes the routine's parameters onto the stack (or stores them into registers) and then jumps to the routine or calls a trap that internally jumps to the routine. If the routine exists as classic 68K code, no mode switch is required and the routine is called as usual. If, however, the routine exists as PowerPC code, the calling application must implicitly invoke the Mixed Mode Manager.

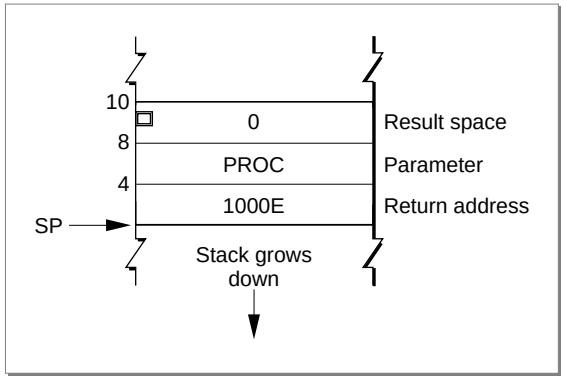
If the calling application merely jumps to the PowerPC code, the code must begin with a routine descriptor, as explained in "Accelerated and Fat Resources," beginning on page 7-4. If the calling application calls a trap, the trap dispatch table must contain—instead of the address of the routine's executable code—the address of a routine descriptor for that routine. This routine descriptor is created at system startup time.

For example, suppose that your application calls the `CountResources` function, as follows:

```
myResCount = CountResources('PROC');
```

Suppose further that `CountResources` has been ported to the PowerPC instruction set. When your application calls `CountResources`, the stack looks like the one shown in Figure 6-2.

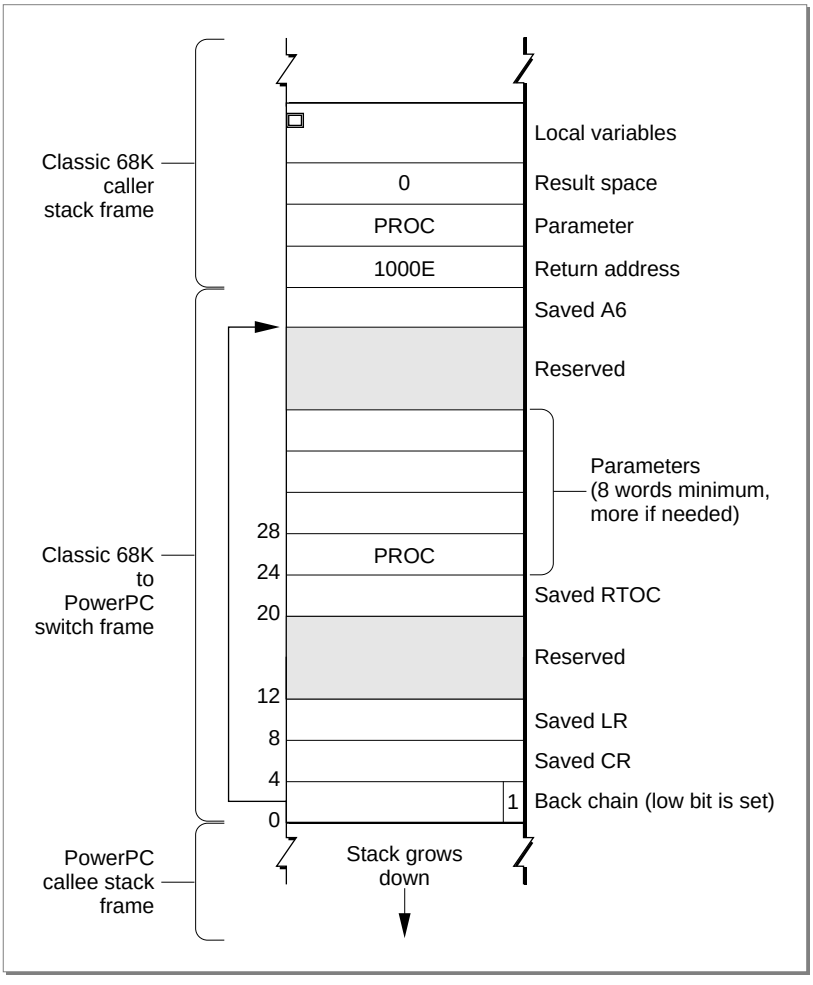
Figure 6-2 The stack before a mode switch



The trap dispatcher executes the `CountResources` routine descriptor, which begins with an executable instruction that invokes the Mixed Mode Manager. The Mixed Mode Manager retrieves the transition vector and creates a switch frame on the stack. A **switch frame** is a stack frame that contains information about the routine to be executed, the state of various registers, and the address of the previous frame. Figure 6-3 shows the structure of a classic 68K to PowerPC switch frame.

Note
In Figure 6-3 the low bit in the back chain pointer to the saved A6 value is set. This bit signals to the Mixed Mode Manager that a switch frame is on the stack. The Mixed Mode Manager fails if the stack pointer has an odd value. ◆

Figure 6-3 A classic 68K to PowerPC switch frame



In addition to creating a switch frame, the Mixed Mode Manager also sets up several CPU registers:

- The PowerPC base register (GPR2) must be set to the direct data area of the fragment containing the `CountResources` routine. This value is obtained from the transition vector whose address is extracted from the routine descriptor.
- The Link Register (LR) must be set to point to code that cleans up the stack and restarts the emulator.

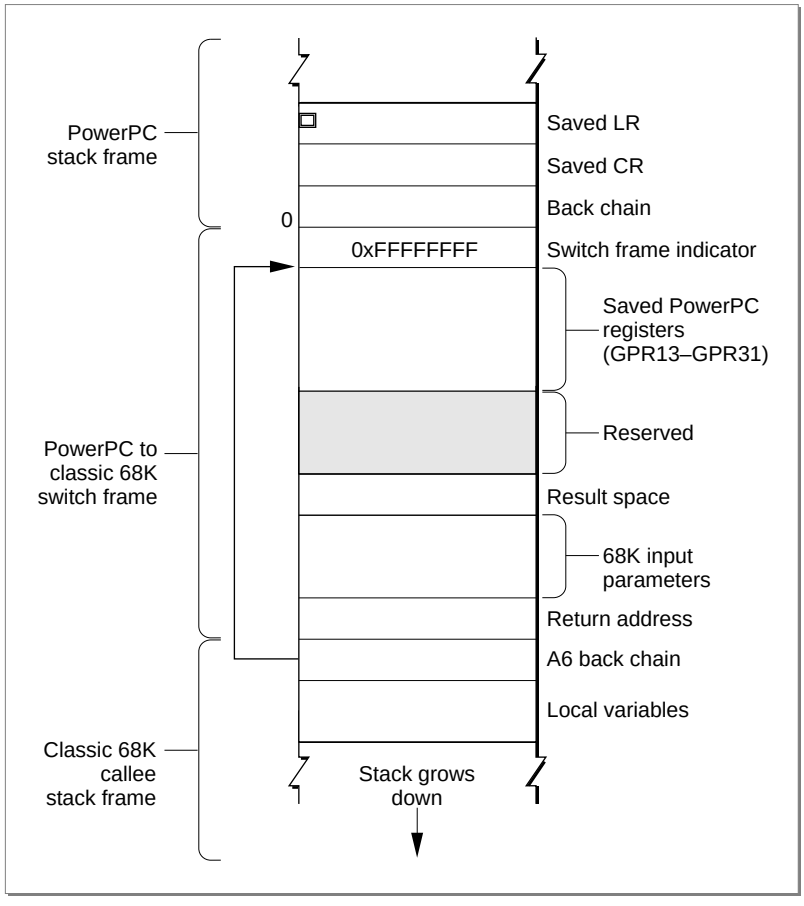
At this point, it's safe to execute the native `CountResources` code. When `CountResources` completes, the Mixed Mode Manager pops the return address and parameters off the stack (since `CountResources` follows Pascal calling conventions). The GPR2, LR, and CR are restored to their saved values, and the switch frame is popped off the stack. The Mixed Mode Manager then jumps back into the 68LC040 Emulator, and the application continues execution.

Calling Classic 68K Code From PowerPC Code

This section describes how the Mixed Mode Manager switches modes from the PowerPC native environment to the classic 68K emulated environment. When PowerPC code calls classic 68K code, the call must go through the routine `CallUniversalProc`.

The call to `CallUniversalProc` invokes the Mixed Mode Manager, which verifies that a mode switch is necessary. At that point, the Mixed Mode Manager saves all nonvolatile registers and other necessary information on the stack in a switch frame. Figure 6-4 shows the structure of a PowerPC to classic 68K switch frame.

Figure 6-4 A PowerPC to classic 68K switch frame



Once the switch frame is set up, the Mixed Mode Manager sets up the 68LC040 Emulator's context block and then jumps into the emulator. When the routine has finished executing, it attempts to jump to the return address pushed onto the stack. That return address points to a "return-to-native" signal (currently stored in the reserved area of the stack) that is used by the Mixed Mode

Manager and the emulator to transfer back to PowerPC code. Once this is done, the Mixed Mode Manager restores native registers that were previously saved and deallocates the switch frame. Control then returns to the caller of `CallUniversalProc`.

IMPORTANT

As currently implemented, the instruction that causes a return from the 68LC040 Emulator to the native PowerPC environment clears the low-order 5 bits of the Condition Code Register (CCR). This prevents 68K callback procedures from returning information in the CCR. If you want to port classic 68K code that calls an external routine that returns results in the CCR, you must instead call a classic 68K stub that saves that information in some other place. ▲

Calling CFM-68K Code From Classic 68K Code

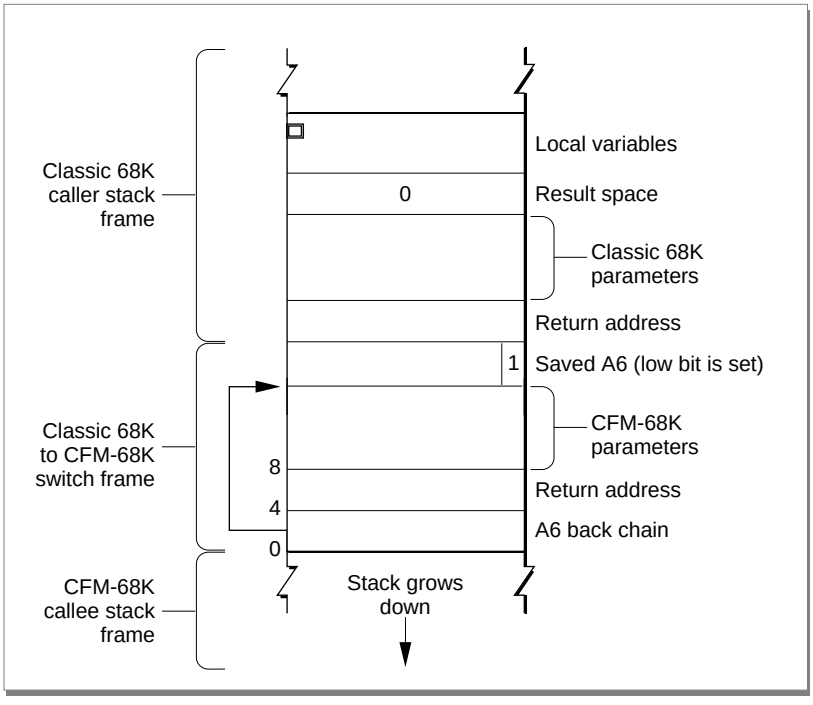
Calling CFM-68K code from a classic 68K routine is very similar to calling PowerPC code from emulated classic 68K code. However, since no virtual machine switch is needed, the switch frame is simpler.

When the Mixed Mode Manager is invoked through the trap in the routine descriptor, it sets up a classic 68K to CFM-68K switch frame before calling the CFM-68K routine. Figure 6-5 shows the switch frame.

Note

The low bit of the saved A6 register is set to indicate that a switch frame is on the stack. This is analogous to the set low bit of the back frame in the classic 68K to PowerPC switch frame. ◆

Figure 6-5 A classic 68K to CFM-68K switch frame

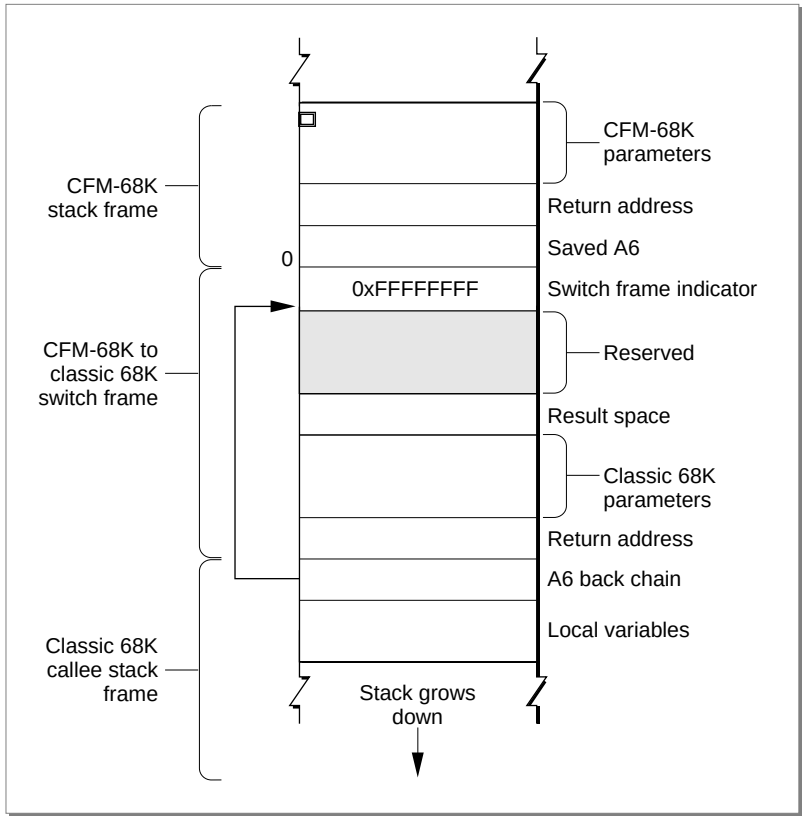


After returning from the called routine, the Mixed Mode Manager copies the return value to its proper location (in a register or on the stack) and pops the stack frame and return address off the stack. If the calling routine uses Pascal calling conventions, the calling routine's parameters are also popped off the stack. Control then passes back to the classic 68K code.

Calling Classic 68K Code From CFM-68K Code

Calling classic 68K code from CFM-68K code is analogous to calling classic 68K code from PowerPC code. The call to `CallUniversalProc` invokes the Mixed Mode Manager, which verifies that a mode switch is necessary. The Mixed Mode Manager sets up a CFM-68K to classic 68K switch frame before calling the classic 68K code. Figure 6-6 shows the structure of the switch frame.

Figure 6-6 A CFM-68K to classic 68K switch frame



After returning from the call, the return value is copied to register D0, and the switch frame is popped off the stack. Control then passes back to the CFM-68K code.

CHAPTER 7

Fat Binary Programs

Contents

A **fat binary program** (often simply called a *fat program*) is an program that contains executable code for more than one runtime architecture. Typically you create fat programs when you want compatibility between hardware platforms. For example, a fat application with PowerPC code and CFM-68K code can be executed on both PowerPC-based and 68K-based Macintosh computers. A user can store a fat application on a portable hard drive and then move it between a 68K-based computer and a PowerPC- based computer.

In addition, fat programs designed to patch code or resources can improve performance by reducing the work of the Mixed Mode Manager. Since a fat program can contain both CFM-based code and classic 68K code, it can execute whichever code is necessary to avoid a mode switch.

Creating Fat Binary Programs

There are two primary reasons for building fat programs:

- You would like to build an application that runs on both PowerPC-based and 68K-based machines. Fat programs of this type contain PowerPC code and either CFM-68K or classic 68K code.
- You are building a shared library, and you would like to have one library file that runs on both PowerPC and 68K-based machines. Fat programs of this type contain PowerPC code and CFM-68K code.

It is very easy to build a fat application to run PowerPC code and classic 68K code, because you can place the PowerPC code in the data fork of an application file, and the classic 68K code can be placed in `'CODE'` resources in the resource fork of the same file. If the application is launched on a PowerPC-based Macintosh computer, the Process Manager recognizes the `'cfrg'0` resource and knows to execute the PowerPC fragment in the data fork of the file. The PowerPC Process Manager simply ignores the `'CODE'` resources in the resource fork. If the application is launched on a 68K-based Macintosh computer, the Process Manager ignores the `'cfrg'0` resource and executes the `'CODE'` resources in the resource fork.

To create a fat application that contains both PowerPC code and CFM-68K code is slightly more complicated, because both PowerPC and CFM-68K runtime fragments require a `'cfrg'0` resource.

IMPORTANT
If you want to run CFM-based code on both PowerPC-based and 68K-based Macintosh computers, you must build the fragment as a fat program. ▲

As described in "The Code Fragment Resource," beginning on page 1-25, the `'cfrg'0` resource is an array that identifies, among other things, the instruction set, type, and name of the fragment. It also identifies the location of the PEF container for the fragment. As an array, the `'cfrg'0` resource can point to multiple containers.

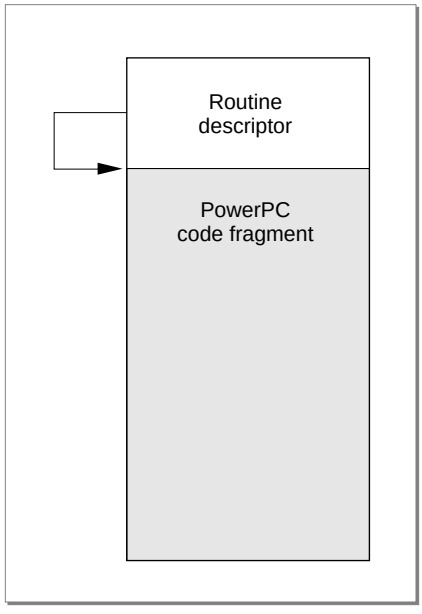
A fat application for both PowerPC and CFM-68K runtime architectures contains code for each instruction set and a `'cfrg'0` resource array that points to both containers. If you launch your application on a 68K Macintosh, the 68K Process Manager reads the `'cfrg'0` resource and launches the CFM-68K runtime version of the application. If you launch your application on a PowerPC-based Macintosh, the PowerPC Process Manager reads the `'cfrg'0` resource and launches the PowerPC version of the application.

You can also create a fat shared library containing both a CFM-68K runtime library and a PowerPC runtime library. Both the PowerPC and CFM-68K fragments are stored in the data fork, with the `'cfrg'0` resource pointing to their locations. In this way, you can ship a single library file that supports both CFM-68K and PowerPC runtime applications.

Accelerated and Fat Resources

In some cases you may want to put an executable CFM-based code fragment into a resource to obtain a CFM-based version of a classic 68K stand-alone code module. For example, you might recompile an existing Hypercard XCMD (eXternal CoMmanD) procedure (which is stored in a resource of type `'XCMD'`) into PowerPC code. However, because the Hypercard application that calls your XCMD procedure could be classic 68K code, a mode switch to the PowerPC environment might be required before your definition procedure can be executed. As a result, you need to add a routine descriptor at the beginning of the resource, as shown in Figure 7-1. These kinds of resources are called **accelerated resources** because they are faster implementations of their classic 68K counterparts. You can transparently replace classic 68K code resources with accelerated PowerPC code resources without having to change the software (for example, an application) that uses them.

Figure 7-1 The structure of an accelerated resource



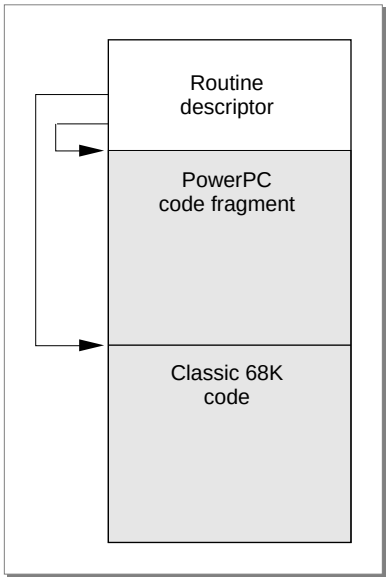
IMPORTANT
Storing CFM-based code in resources is generally not recommended and should be done only in cases where you have no control over the code that calls it. If you are designing a plug-in interface, the plug-ins should be stored in the data fork. ▲

The routine descriptor is necessary for the Mixed Mode Manager to know whether it needs to change modes in order to execute the code. The routine descriptor also lets the Mixed Mode Manager know whether it needs to call the Code Fragment Manager to prepare the fragment.

The `procDescriptor` field of the routine record—contained in the `routineRecords` field of the routine descriptor—should contain the offset from the beginning of the resource (that is, the beginning of the routine descriptor) to the beginning of the executable code fragment. In addition, the routine flags for the specified code should have the `kProcDescriptorIsRelative` bit set, indicating that the address is relative, not absolute. If the code contained in the resource is CFM-based code, you should also set the `kFragmentNeedsPreparing` bit.

You can also create **fat resources**, that is, resources containing both classic 68K and PowerPC versions of some routine. Figure 7-2 shows the general structure of such a resource.

Figure 7-2 The structure of a fat resource



In this case, the routine descriptor contains two routine records in its `routineRecords` field, one describing the classic 68K code and one describing the PowerPC code. As with any code-bearing resource, the `procDescriptor` field of each routine record should contain the offset from the beginning of the resource to the beginning of the appropriate code. The flags for both routine records should have the `kProcDescriptorIsRelative` flag set, and the routine flags for the PowerPC routine record should have the `kFragmentNeedsPreparing` flag set.

Note
You can also create a fat resource that contains CFM-68K code and classic 68K code, although there are no obvious advantages. However, doing so may simplify static data access or code compatibility in some cases. ◆

Since a fat resource contains a routine descriptor at its entry point, it assumes that the host system contains the Mixed Mode Manager. If this is not the case, a problem can arise when the Mixed Mode trap is invoked. A solution is to create a variant called a **safe fat resource,** which begins with extra classic 68K code to check for the presence of the Mixed Mode Manager. If the Mixed Mode Manager is present, the code should move a routine descriptor to the beginning of the resource. If the Mixed Mode Manager is not present, it should add a branch instruction at the beginning to jump directly to the classic 68K portion of the resource. Thus the first call to the resource uses a few extra instruction cycles, but subsequent calls are faster.

Note
In MPW, the interface file `MixedMode.r` provides Rez templates that you can use to create the accelerated resource shown in Figure 7-1 or the fat resource shown in Figure 7-2. The file also contains sample code for creating a safe fat resource. ◆

▲ **WARNING**
Do not call accelerated resources at interrupt time. If the resource containing the code has not been prepared, the Code Fragment Manager will be called to do so, and the Code Fragment Manager cannot run at interrupt time. ▲

Sometimes it's useful to keep the executable code of a definition function in some location other than a resource. To do this, you need to create a stub definition resource that is of the type expected by the system software and that simply jumps to your code. For example, Listing 7-1 shows the Rez input for a stub list definition resource.

Listing 7-1 Rez input for a stub list definition resource

```
data 'LDEF' (128, "MyCustomLDEF", preload, locked) {
    /*need to fill in destination address before using this stub*/
    $"41FA 0006"    /*LEA PC+8, A0  ;A0 <- ptr to destination address*/
    $"2050"         /*MOVEA.L (A0), A0;A0 <- destination address*/
    $"4ED0"             /*JMP (A0)      ;jump to destination address*/
    $"00000000"         /*destination address*/
};
```

Your application (or other software) is responsible for filling in the destination address before the list definition procedure is called by the List Manager. For classic 68K code, the destination address should be the address of the list definition procedure itself. For PowerPC-based code, the destination address should be a universal procedure pointer (that is, the address of a routine descriptor for the list definition procedure).

It's important to understand the distinction between accelerated resources and a normal resource-based fragment (sometimes called a **private resource**), so that you know when to create them and how to load and execute the code they contain. An accelerated resource is any resource containing PowerPC code that has a single entry point at the top (the routine descriptor) and that models the traditional behavior of a classic 68K stand-alone code resource. There are many examples, including menu definition procedures (stored in resources of type `'MDEF'`), control definition functions (stored in resources of type `'CDEF'`), window definition functions (stored in resources of type `'WDEF'`), list definition procedures (stored in resources of type `'LDEF'`), HyperCard extensions (stored in resources of type `'XCMD'`), and so forth. A private resource is any other kind of executable resource whose code is called directly by your application.

In most cases, you don't need to do anything special to get the system software to recognize your accelerated resource and to call it at the appropriate time. For example, the Menu Manager automatically loads a custom menu definition procedure into memory when you call `GetMenu` for a menu whose `'MENU'` resource specifies that menu definition procedure. Similarly, HyperCard calls code like that shown in Listing 7-2 to load a resource of type `'XCMD'` into memory and execute the code it contains.

Listing 7-2 Using an accelerated resource

```
Handle          myHandle;
XCmdBlock       myParamBlock;

myHandle = Get1NamedResource('XCMD', '\pMyXCMD');
HLock(myHandle);

/*Fill in the fields of myParamBlock here.*/

CallXCMD(&myParamBlock, myHandle);
HUnlock(myHandle);
```

The caller of an accelerated resource executes the code either by jumping to the code (if the caller is classic 68K code) or by calling the Mixed Mode Manager `CallUniversalProc` function (if the caller is PowerPC code). In either case, the Mixed Mode Manager calls the Code Fragment Manager to prepare the fragment, which is already loaded into memory. With accelerated resources, you don't need to call the Code Fragment Manager yourself. In fact, you don't need to do anything special at all for the system software to recognize and use your accelerated resource if you've built it correctly. This is because the system software is designed to look for, load, and execute those resources in the appropriate circumstances. In many cases, your application passes to the system software just a resource type and resource ID. The resource must begin with a routine descriptor, so that the dereferenced handle to the resource is a universal procedure pointer.

The code shown in Listing 7-2 (or similar code for any other accelerated resource) can be executed multiple times with no appreciable performance loss. If the code resource remains in memory, the only overhead incurred by Listing 7-2 is to lock the code, fill in the parameter block, jump to the code, and then unlock it. However, because of the way in which the system software manages your accelerated resources, there are several important restrictions on their operation:

- An accelerated resource cannot contain a termination routine, largely because the Code Fragment Manager does not know when the resource is released. The Code Fragment Manager effectively forgets about the connection to your resource as soon as it has prepared the resource for execution.

- An accelerated resource must contain a main symbol, which must be a procedure. For example, in an accelerated `'MDEF'` resource, the main procedure must be the menu definition procedure itself (which typically dispatches to other routines contained in the resource).
- You cannot call the Code Fragment Manager routine `FindSymbol` to get information about the exported symbols in an accelerated resource. More generally, you cannot call any Code Fragment Manager routine that requires a connection ID as a parameter.
- The fragment's data section is instantiated in place (that is, within the block of memory into which the resource itself is loaded). For in-place instantiation, you need to build an accelerated resource using an option that specifies that the data section of the fragment not be compressed. See the documentation for your software development system to determine how to specify uncompressed data sections.
- Accelerated resources can move in memory or be purged like classic 68K resources (note that the code in Listing 7-2 unlocks the `'XCMD'` resource after executing it). If the resource moves between calls, some of the global data in the resource might become invalid. For example, a global pointer may end up dangling if the code or data it points to has moved.

To allow accelerated PowerPC resources to be manipulated just like classic 68K code resources, the Mixed Mode Manager and the Code Fragment Manager cooperate to make sure that the code is ready to be executed when it is called. If the resource code hasn't been moved since it was prepared for execution, then no further action is necessary. If, however, the code resource has moved or been reloaded elsewhere in memory, some of the global data in the resource might have become invalid. To help avoid dangling pointers, the Code Fragment Manager always updates any pointers in the fragment's data section that are initialized at compile time and not modified at runtime.

IMPORTANT
The Code Fragment Manager cannot update all global data references in an accelerated resource that has moved in memory. Therefore, an accelerated resource must not use global pointers (in C code, pointers declared as `extern` or `static`) that are either initialized at runtime or contained in dynamically allocated data structures to point to code or data contained in the resource itself. An accelerated resource can use uninitialized global data to point to objects in the heap. In addition, an accelerated resource can use global pointers that are initialized at compile time to point to functions, other global data, and literal strings, but these pointers cannot be modified at runtime. ▲

The best way to avoid the global data restrictions on an accelerated resource is to put the global data used by the accelerated resource into an import library. Since the accelerated resource is a fragment, it can import both code and data from the library. The import library's code and data are fixed in memory, and the library is unloaded only when your application terminates, not when the accelerated resource is purged.

If you must declare global variables in your accelerated resource, you should check Listing 7-3 for examples of acceptable declarations. Note that these declarations assume the resource code does not change the values of the initialized variables.

Listing 7-3 Acceptable global declarations in an accelerated resource

```
int a;          /*uninitialized; not modified if resource moves*/
Ptr myPtr;      /*uninitialized; not modified if resource moves; */
                /* can be assigned at runtime to point to heap object*/
Handle *h;      /*uninitialized; not modified if resource moves; */
                /* can be assigned at runtime to point to heap object*/
int *b = &a;                        /*updated each time resource moves*/
char *myStr = "Hello, world!";      /*updated each time resource moves*/
extern int myProcA(), myProcB();
struct {
    int     (*one)();
    int     (*two)();
    char    *str;
} myRec = {myProcA, myProcB, "Hello again!"};
            /*all three pointers are updated each time resource moves*/
```

Listing 7-4 shows examples of data declarations and code that do not work in an accelerated resource that is moved or purged.

Listing 7-4 Unacceptable global declarations and code in an accelerated resource

```
int a;
int *b;
int *c = &a;
Ptr (*myPtr) (long) = NewPtr;
static Ptr MyNewPtr();
struct myHeapStruct {
    int         *b;
    Ptr         (myPtr) (long);
} *hs;

b = &a;             /*b does not contain &a after resource is moved*/
c = NULL;           /*c does not contain NULL after resource is moved*/
c = (int *) NewPtr(4);      /*dangling pointer after resource is moved*/
myPtr = MyNewPtr;           /*dangling pointer after resource is moved*/
hs = NewPtr(sizeof(myHeapStruct));
            /*hs still points to nonrelocatable heap block after move*/
hs->b = &a;         /*hs->b will not point to global a after move*/
hs->myPtr = MyNewPtr;
                    /*hs->myPtr will not point to MyNewPtr after move*/
```

PEF Structure

Contents

This chapter describes the structure of the PEF storage standard, which is the format used to store programs in the Code Fragment Manager–based runtime architecture. You need this information if you read from or write to PEF containers—if you are writing a compiler or other development tool, for example.

After a high-level view of a PEF container in the "Overview" section, there follow sections that describe the elements of a PEF container in more detail.

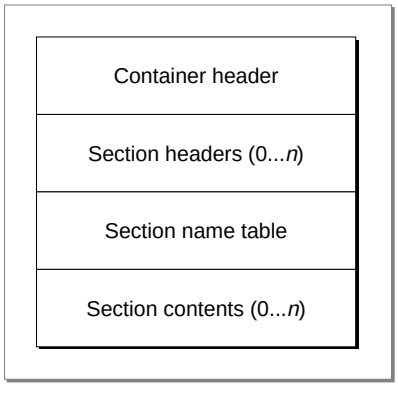
Note that the PEF storage standard is not exclusive to the CFM-based architecture. Other architectures can follow the specification described here and use PEF containers to store their code and data. In such cases, the appropriate PEF handler for that architecture takes the role of the Code Fragment Manager.

Overview

The CFM-based architecture stores information in **PEF containers,** which are simply storage blocks that contain PEF information. A PEF container can be stored in a file, a resource, or section of memory. The Code Fragment Manager can transparently prepare any of these forms.

A PEF container has four major parts as shown in Figure 8-1.

Figure 8-1 Structure of a PEF container



The four parts are as follows:

- The **container header** contains information about the container itself, such as the runtime architecture that it was created for, version information, and so on.
- Each **section header** contains information (size, alignment, and so on) about the various **sections** in the PEF container. Both code and data can be stored in sections.
- The section name table contains the names of each section.
- The section contents area contains the contents of the sections described by the section headers.

PEF containers typically include one or more sections of executable code, one or more sections of initialized data, and a loader section.

Each part is described in more detail in the sections that follow.

The Container Header

The container header contains information about the specific PEF container. The container header data structure is of fixed size (40 bytes) and has the form shown in Listing 8-1.

Listing 8-1 PEF container header data structure

```
struct PEFContainerHeader {
    OSType  tag1;
    OSType  tag2;
    OSType  architecture;
    UInt32  formatVersion;
    UInt32  dateTimeStamp;
    UInt32  oldDefVersion;
    UInt32  oldImpVersion;
    UInt32  currentVersion;
    UInt16  sectionCount;
    UInt16  instSectionCount;
    UInt32  reservedA;
};
```

The fields in the container header are as follows:

- The `tag1` field (4 bytes) designates that the container uses an Apple-defined format. This field must be set to `Joy!` in ASCII.
- The `tag2` field (4 bytes) identifies the type of container (currently set to `peff` in ASCII).
- The `architecture` field (4 bytes) indicates the architecture type that the container was generated for. This field holds the ASCII value `pwpc` for the PowerPC CFM implementation or `m68k` for CFM-68K.
- The `formatVersion` field (4 bytes) indicates the version of PEF used in the container. The current version is `1`.
- The `dateTimeStamp` field (4 bytes) indicates when the PEF container was created. The stamp follows the Macintosh time-measurement scheme (that is, the number of seconds measured from January 1, 1904).
- The next three fields, `oldDefVersion`, `oldImpVersion`, and `currentVersion` (4 bytes each), contain version information that the Code Fragment Manager uses to check shared library compatibility. For more information about version checking, see "Checking for Compatible Import Libraries" (page 1-19).
- The `sectionCount` field (2 bytes) indicates the total number of sections contained in the container.
- The `instSectionCount` field (2 bytes) indicates the number of instantiated sections. Instantiated sections contain code or data that are required for execution.
- The `reservedA` field (4 bytes) is currently reserved and must be set to `0`.

PEF Sections

A PEF container can contain any number of sections. A section usually contains code or data. A special case is the loader section, which is discussed separately in "The Loader Section" (page 8-15). For each section there is a header, which includes information such as the type of section, its presumed runtime address, its size, and so on, and a corresponding section contents area.

Sections are numbered from 0, based on the position of their header, and the sections are identified by these numbers. However, the corresponding section

contents do not have to be in the same order as the section headers. The only requirement is that instantiated section headers (that is, headers for sections containing code or data) must precede noninstantiated ones in the section header array.

The section header data structure is of fixed size (28 bytes) and has the form shown in Listing 8-2.

Listing 8-2 Section header data structure

```
struct PEFSectionHeader {
    SInt32  nameOffset;
    UInt32  defaultAddress;
    UInt32  totalSize;
    UInt32  unpackedSize;
    UInt32  packedSize;
    UInt32  containerOffset;
    UInt8   sectionKind;
    UInt8   shareKind;
    UInt8   alignment;
    UInt8   reservedA;
};
```

The fields in the section header are as follows:

- The `nameOffset` field (4 bytes) holds the offset from the start of the section name table to the location of the section name. The name of the section is stored as a C-style null-terminated character string.

  If the section has no name, the `nameOffset` field contains `-1`.

- The `defaultAddress` field (4 bytes) indicates the preferred address (as designated by the linker) at which to place the section's instance. If the Code Fragment Manager can place the instance in the preferred memory location, the load-time and link-time addresses are identical and no internal relocations need to be performed.

- The `totalSize` field (4 bytes) indicates the size, in bytes, required by the section's contents at execution time. For a code section, this size is merely the size of the executable code. For a data section, this size indicates the sum of the size of the initialized data plus the size of any zero-initialized data.

Zero-initialized data appears at the end of a section's contents and its length is exactly the difference of the `totalSize` and `unpackedSize` values.

For noninstantiated sections, this field is ignored.

- The `unpackedSize` (4 bytes) is the size of the section's contents that is explicitly initialized from the container. For code sections, this field is the size of the executable code. For an unpacked data section, this field indicates only the size of the initialized data. For packed data this is the size to which the compressed contents expand. The `unpackedSize` value also defines the boundary between the explicitly initialized portion and the zero-initialized portion.

  For noninstantiated sections, this field is ignored.

- The `packedSize` field (4 bytes) indicates the size, in bytes, of a section's contents in the container. For code sections, this field is the size of the executable code. For an unpacked data section, this field indicates only the size of the initialized data. For a packed data section (see Table 8-1 (page 8-8)) this field is the size of the pattern description contained in the section.

- The `containerOffset` field (4 bytes) contains the offset from the beginning of the container to the start of the section's contents. Packed data sections and the loader section should be 4-byte aligned. Code sections and data sections that are not packed should be at least 16-byte aligned.

- The `sectionKind` field (1 byte) indicates the type of section as well as any special attributes. Table 8-1 (page 8-8) shows the currently supported section types. Note that instantiated read-only sections cannot have zero-initialized extensions.

- The `shareKind` field (1 byte) controls how the section information is shared among processes by the Code Fragment Manager. You can specify any of the sharing options shown in Table 8-2 (page 8-9).

- The `alignment` field (1 byte) indicates the desired alignment for instantiated sections in memory as a power of 2. A value of `0` indicates 1-byte alignment, `1` indicates 2-byte (halfword) alignment, `2` indicates 4-byte (word) alignment, and so on. Note that this field does not indicate the alignment of raw data relative to a container. The Code Fragment Manager does not support this field under System 7.

  In System 7, the Code Fragment Manager gives 16-byte alignment to all writable sections. The alignment of read-only sections, which are used directly from the container, is dependent on the alignment of the section's

contents within the container and the overall alignment of the container itself. When the container is not file-mapped, the overall container alignment is 16 bytes. When the container is file-mapped, the entire data fork is mapped and aligned to a 4KB boundary. The overall alignment of a file-mapped container thus depends on the container's alignment within the data fork. Note that file-mapping is currently supported only on PowerPC machines, and only when virtual memory is enabled.

- The `reservedA` field (1 byte) is currently reserved and must be set to `0`.

Table 8-1 shows the various types of sections that can appear in PEF containers and the corresponding value in the `sectionKind` field.

Table 8-1 Section types

Value	Type	Instantiated?	Description
`0`	Code	Yes	Contains read-only executable code in an uncompressed binary format. A container can have any number of code sections. Code sections are always shared.
`1`	Unpacked data	Yes	Contains uncompressed, initialized, read/write data followed by zero-initialized read/write data. A container can have any number of data sections, each with a different sharing option.
`2`	Pattern-initialized data	Yes	Contains read/write data initialized by a pattern specification contained in the section's contents. The contents essentially contain a small program that tells the Code Fragment Manager how to initialize the raw data in memory. A container can have any number of pattern-initialized data sections, each with its own sharing option. See "Pattern-Initialized Data" (page 8-10) for more information about creating pattern specifications.
`3`	Constant	Yes	Contains uncompressed, initialized, read-only data. A container can have any number of constant sections, and they are implicitly shared.

continued

Table 8-1 Section types (continued)

Value	Type	Instantiated?	Description
4	Loader	No	Contains information about imports, exports, and entry points. See "The Loader Section" (page 8-15) for more details. A container can have only one loader section.
5	Debug	N/A	Reserved for future use.
6	Executable data	Yes	Contains information that is both executable and modifiable. For example, this section can store code that contains embedded data. A container can have any number of executable data sections, each with a different sharing option.
7	Exception	N/A	Reserved for future use.
8	Traceback	N/A	Reserved for future use.

Table 8-2 shows the sharing options available for PEF sections and the corresponding value in the `shareKind` field.

Table 8-2 Sharing options

Type	Value	Description
Process share	1	Indicates that the section is shared within a process, but a fresh copy is created for different processes.
Global share	4	Indicates that the section is shared between all processes in the system.
Protected share	5	Indicates that the section is shared between all processes, but is protected. Protected sections are read/write in privileged mode and read-only in user mode. This option is not available in System 7.

The Section Name Table

The PEF container section name table contains the names of the sections stored as C-style null-terminated character strings. The strings have no specified alignment. Note that the section name table must immediately follow the section headers in the container.

Section Contents

The contents of a PEF section varies depending on the section type. For code and unpacked data sections, the section contains the executable code or initialized data as they would appear when loaded into memory. For some other sections, the raw section data must be manipulated by the Code Fragment Manager before loading. For example, a pattern-initialized data section does not contain simple data, but rather it contains a pattern specification that tells the loader how to initialize the section.

Section data within a container must be at least 16-byte aligned if the section type is instantiated and directly usable (code or data, for example, but not pattern-initialized). Noninstantiated sections should be at least 4-byte aligned. Note that gaps may appear between sections due to alignment restrictions; you cannot be sure that adding the offset of a section to its length will locate the beginning of the next section.

Pattern-Initialized Data

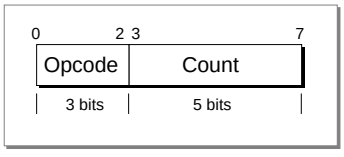
Because the data stored in a PEF container acts only as a template for the instantiated version of the data section at runtime, it is preferable to compact the stored data section. Pattern-initialized data (pidata) allows you to replace repetitious patterns of data (for example, in transition vector arrays and C++ VTables) with small instructions that generate the same result. These instructions save space (resulting in a data section about one third the size of a similar uncompressed one) and can be executed quickly at preparation time.

Note
The choice of data generation patterns reflects the code generation model used to build CFM-based runtime fragments. ◆

To execute the pattern-initialization instructions, a data location counter must be set to the first byte of the data section in memory and an instruction location counter must be set to the first byte of the pattern-initialized data. Each opcode instruction (and its associated arguments) is executed in turn until the end of the pattern-initialized data section is reached. The data location counter is incremented each time a data byte is written.

Figure 8-2 shows the general format of a pattern-initialization instruction.

Figure 8-2 A pattern-initialization instruction

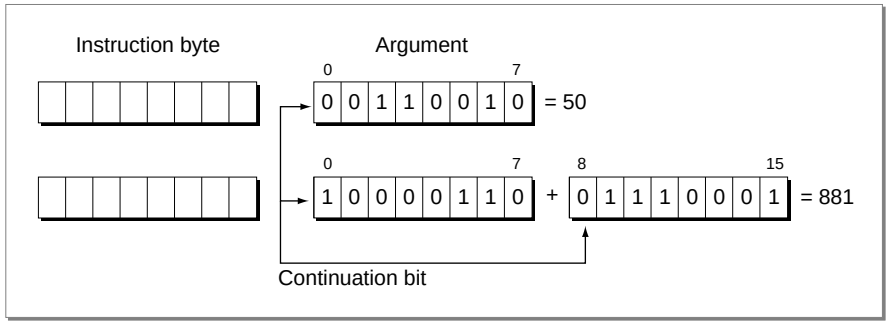


Each instruction, depending on its definition, takes one or more arguments. The first is stored in the 5 bits of the count field while any additional arguments are stored in bytes that immediately follow the instruction byte. Each instruction may also require raw data used in the initialization process; this raw data appears after the argument bytes.

The instruction byte can hold count values up to 31. If you need to specify a count value larger than 31, you should place 0 in the count field. This indicates that the first argument following the instruction byte is the count value.

Argument values are stored in big-endian fashion, with the most significant bits first. Each byte holds 7 bits of the argument value. The high-order bit is set for every byte except the last (that is, an unset high-order bit indicates the last byte in the argument). For example, Figure 8-3 shows how the values 50 and 881 would be stored.

Figure 8-3 Argument storage in pattern-initialized data



The argument value is determined by shifting the current value up 7 bits and adding in the low-order 7 bits of the next byte, doing so until an unset high-order bit is encountered.

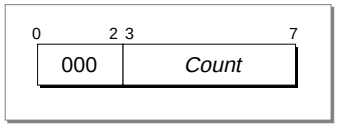
You can encode up to a 32-bit value using this format. In the case of a 32-bit value, the fifth byte must have 0 in its high-order bit, and only the least-significant 32 bits of the 35-bit accumulation are used.

Note
The advantage of this format is that while a 32-bit value is stored in 5 bytes, smaller values can be stored in correspondingly fewer bytes. ◆

Pattern-Initialization Opcodes

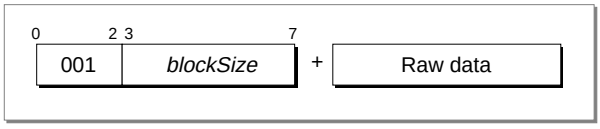
The sections that follow describe the currently defined pattern-initialization instructions. Opcodes 101, 110, and 111 are reserved for future use.

Zero (Opcode 000)



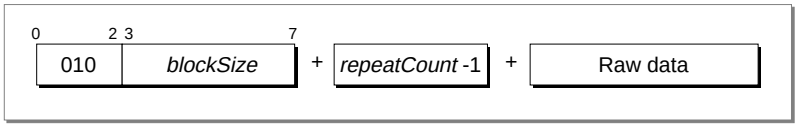
This instruction initializes *Count* bytes to 0 beginning at the current data location.

blockCopy (Opcode 001)



This instruction initializes the next *blockSize* bytes from the current data location to the values in the following raw data bytes.

repeatedBlock (Opcode 010)

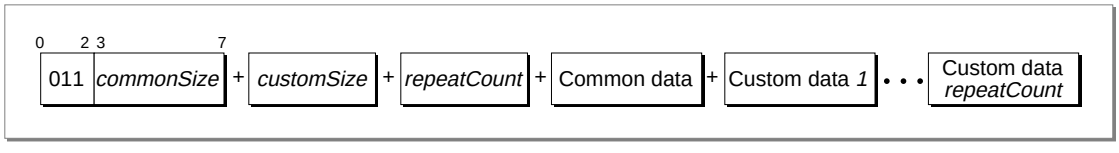


This instruction repeats the *blockSize* number of data bytes *repeatCount* times, beginning at the current data location.

IMPORTANT

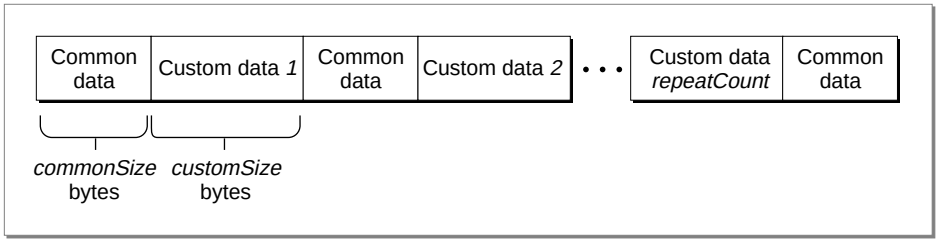
The repeat count value stored in the instruction is one smaller than the actual value (*repeatCount* -1). ▲

interleaveRepeatBlockWithBlockCopy (Opcode 011)

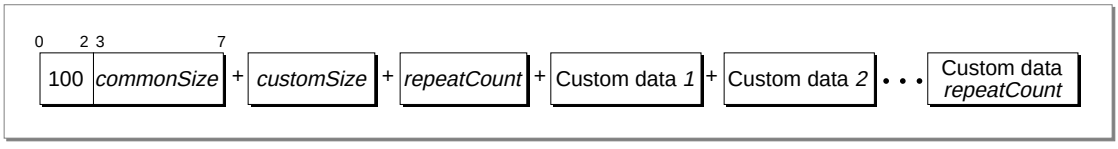


This instruction requires three parameters and *commonSize* + (*customSize* * *repeatCount)* bytes of raw data. The first *commonSize* bytes of raw data make up the common (repeating) pattern and the next *customSize* bytes make up the first custom (nonrepeating) section. There are *repeatCount* number of custom sections. The instruction places the common pattern followed by the first custom section, then the common pattern, then the second custom section, and so on. After performing this procedure *repeatCount* times, a final common data pattern is added at the end. Figure 8-4 shows the data section after initialization.

Figure 8-4 Data section after executing `interleaveRepeatBlockWithBlockCopy`

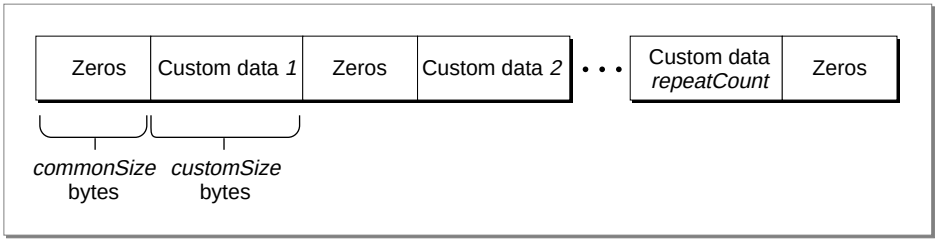


interleaveRepeatBlockWithZero (Opcode 100)



This instruction is similar to the `interleaveRepeatBlockWithBlockCopy` instruction except the common pattern is *commonSize* bytes of zero instead of raw data. Figure 8-5 shows the data section after initialization.

Figure 8-5 Data section after executing `interleaveRepeatBlockWithZero`

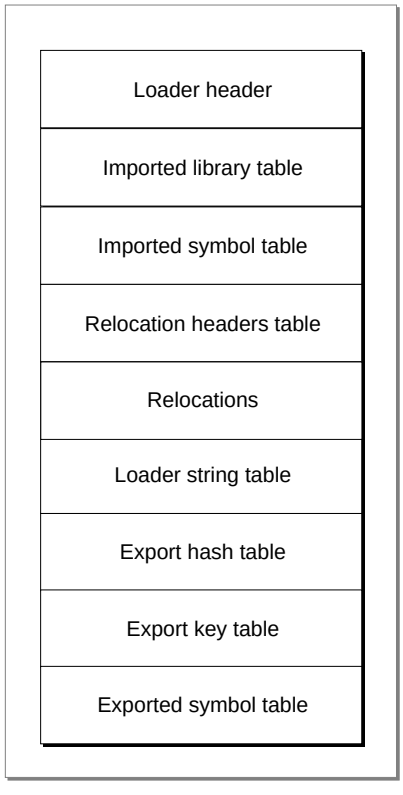


The Loader Section

The loader section is a special section that contains information used by the Code Fragment Manager to prepare the fragment. It contains information about the symbols imported to, and exported from, the fragment as well as instructions that tell the Code Fragment Manager how to fix up references to symbols.

The general layout and content of the loader section appears in Figure 8-6.

Figure 8-6 PEF loader section



The contents of the loader section are as follows:

- The loader header contains information about the location of other components of the loader section.
- The import information (library descriptions and symbol tables) describes the imports for the container.
- The relocation headers table provides information about relocations to be applied to a given section.
- The relocations area contains relocation instructions that describe how to fix up references to symbols within each section.
- The loader string table contains the names of the container's imported and exported symbols.
- The export information is contained in a hashed data structure, which has three parts:
  - The export hash table, which contains hash chain information (the number of elements in the chain and the location of the first element) for each index value in the table.
  - The export key table, which contains the hash values of the exports.
  - The exported symbol table, which contains additional information about the exported symbols.

The sections that follow describe these components in more detail.

All tables use zero-based indexes. It is recommended that offset values for elements with no entries be set to `0`.

The Loader Header

The loader header data structure is of fixed size (56 bytes) and has the form shown in Listing 8-3.

Listing 8-3 Loader header data structure

```
struct PEFLoaderInfoHeader {
    SInt32  mainSection;
    UInt32  mainOffset;
    SInt32  initSection;
    UInt32  initOffset;
```

```
    SInt32  termSection;
    UInt32  termOffset;
    UInt32  importedLibraryCount;
    UInt32  totalImportedSymbolCount;
    UInt32  relocSectionCount;
    UInt32  relocInstrOffset;
    UInt32  loaderStringsOffset;
    UInt32  exportHashOffset;
    UInt32  exportHashTablePower;
    UInt32  exportedSymbolCount;
};
```

The fields in the loader header are as follows:

- The `mainSection` field (4 bytes) specifies the number of the section in this container that contains the main symbol. If the fragment does not have a main symbol, this field is set to `-1`.
- The `mainOffset` field (4 bytes) indicates the offset (in bytes) from the beginning of the section to the main symbol.
- The `initSection` field (4 bytes) contains the number of the section containing the initialization function's transition vector. If no initialization function exists, this field is set to `-1`.
- The `initOffset` field (4 bytes) indicates the offset (in bytes) from the beginning of the section to the initialization function's transition vector.
- The `termSection` field (4 bytes) contains the number of the section containing the termination routine's transition vector. If no termination routine exists, this field is set to `-1`.
- The `termOffset` field (4 bytes) indicates the offset (in bytes) from the beginning of the section to the termination routine's transition vector.
- The `importedLibraryCount` field (4 bytes) indicates the number of imported libraries.
- The `totalImportedSymbolCount` field (4 bytes) indicates the total number of imported symbols.
- The `relocSectionCount` field (4 bytes) indicates the number of sections containing load-time relocations.
- The `relocInstrOffset` field (4 bytes) indicates the offset (in bytes) from the beginning of the loader section to the start of the relocations area.

- The `loaderStringsOffset` field (4 bytes) indicates the offset (in bytes) from the beginning of the loader section to the start of the loader string table.
- The `exportHashOffset` field (4 bytes) indicates the offset (in bytes) from the beginning of the loader section to the start of the export hash table. The hash table should be 4-byte aligned with padding added if necessary.
- The `exportHashTablePower` field (4 bytes) indicates the number of hash index values (that is, the number of entries in the hash table). The number of entries is specified as a power of two. For example, a value of `0` indicates one entry, while a value of `2` indicates four entries.

  If no exports exist, the hash table still contains one entry, and the value of this field is `0`.
- The `exportedSymbolCount` field (4 bytes) indicates the number of symbols exported from this container.

Imported Libraries and Symbols

The loader section must describe every import library required by the fragment and the symbols imported from those libraries. The following two sections describe the format of these descriptions.

Imported Library Descriptions

An imported library description, which contains information about a required import library, is of fixed size (24 bytes) and has the form shown in Listing 8-4.

Listing 8-4 Imported library description data structure

```
struct PEFImportedLibrary {
    UInt32  nameOffset;
    UInt32  oldImpVersion;
    UInt32  currentVersion;
    UInt32  importedSymbolCount;
    UInt32  firstImportedSymbol;
    UInt8   options;
    UInt8   reservedA;
    UInt16  reservedB;
};
```

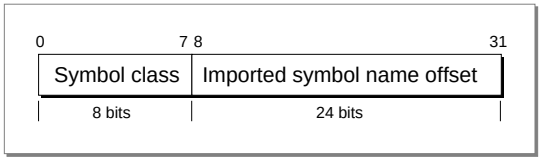
The fields of the description are as follows:

- The `nameOffset` field (4 bytes) indicates the offset (in bytes) from the beginning of the loader string table to the start of the null-terminated library name.
- The `oldImpVersion` and `currentVersion` fields (4 bytes each) provide version information for checking the compatibility of the imported library.
- The `importedSymbolCount` field (4 bytes) indicates the number of symbols imported from this library.
- The `firstImportedSymbol` field (4 bytes) holds the (zero-based) index of the first entry in the imported symbol table for this library.
- The `options` byte contains bit flag information as follows:
  - The high-order bit (mask 0x80) controls the order that the import libraries are initialized. If set to `0`, the default initialization order is used, which specifies that the Code Fragment Manager should *try* to initialize the import library before the fragment that imports it. When set to `1`, the import library *must* be initialized before the client fragment.
  - The next bit (mask 0x40) controls whether the import library is weak. When set to `1` (weak import), the Code Fragment Manager continues preparation of the client fragment (and does not generate an error) even if the import library cannot be found. If the import library is not found, all imported symbols from that library have their addresses set to `0`. You can use this information to determine whether a weak import library is actually present.
- The `reservedA` and `reservedB` fields are currently reserved and must be set to `0`.

The Imported Symbol Table

The imported symbol table is an array of imported symbol entries. Symbols imported from the same library are grouped together in the table, but they may appear in any order within that grouping. A table entry is of fixed size (4 bytes) and has the form shown in Figure 8-7.

Figure 8-7 An imported symbol table entry

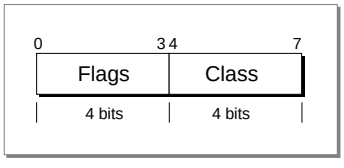


The elements of the table entry are as follows:

- The symbol class field (1 byte) designates the class of the imported symbol.
- The imported symbol name offset field (3 bytes) indicates the offset (in bytes) from the beginning of the loader string table to the null-terminated name of the symbol.

The symbol class byte of an imported symbol entry is structured as shown in Figure 8-8.

Figure 8-8 A symbol class field



For imported symbols, the high-order flag bit (mask 0x80) indicates whether the symbol is weak. When this bit is set, the imported symbol does not have to be present at fragment preparation time in order for execution to continue. However, your code must check that the imported symbol exists before attempting to use it. The other flag bits are currently reserved.

The symbol classes are defined in Table 8-3. The symbol classes are used for annotation only.

Table 8-3 Symbol classes

Class name	Value	Description
`kPEFCodeSymbol`	`0`	A code address
`kPEFDataSymbol`	`1`	A data address
`kPEFTVectSymbol`	`2`	A standard procedure pointer
`kPEFTOCSymbol`	`3`	A direct data area (Table of Contents) symbol
`kPEFGlueSymbol`	`4`	A linker-inserted glue symbol

Relocations

Relocations (sometimes called *fix-ups*) are part of a process by which the Code Fragment Manager replaces references to code and data with actual addresses at runtime. The loader section contains information on how to perform these relocations. These relocations apply to any symbols accessed via pointers, such as imported code and data, or a fragment's own pointer-based function calls.

By the very nature of pointer-based references, you cannot know the actual address that a pointer refers to at build time. Instead, the compiler includes placeholders than can be fixed up by the Code Fragment Manager at preparation time.

For example, a reference to an imported routine points to a transition vector. Before preparation, the pointer in the calling fragment that points to the transition vector has the value `0`. After instantiating the called fragment at preparation time, the actual address of the transition vector becomes known. The Code Fragment Manager then executes a relocation instruction that adds the address of the transition vector to the pointer that references it. The pointer then points to the transition vector in the called fragment's data section.

Relocation information is stored in PEF containers using a number of specialized instructions and variables, which act much like machine-language instructions for a pseudo-microprocessor. These elements reduce the number of bytes required to store the relocation information and reduce the time required to perform the relocations.

The pseudo-microprocessor maintains state information in pseudo-registers. For the state to be correct for each instruction, relocation instructions must be executed in order from start to finish for each section.

The relocation instructions make use of the variables shown in Table 8-4. The initial values are set by the Code Fragment Manager prior to executing the relocations for each section.

Table 8-4 Relocation variables

Name	Description
`relocAddress`	Holds an address within the section where the relocations are to be performed. The initial value is the base address of the section that is to be relocated.
`importIndex`	Holds a symbol index, which is used to access an imported symbol's address.(The address can then be used for relocation.) The initial value is `0`.
`sectionC`	Holds the memory address of an instantiated section within the PEF container; this variable is used by relocation instructions that relocate section addresses. The initial value is the memory address of section 0 if that section is present and instantiated. Otherwise the initial `sectionC` value is `0`. Note that relocation instructions can change the value of `SectionC`; this affects subsequent relocation instructions that refer to this variable. The name `sectionC` is given for convenience only; use of this variable is not restricted to code sections.
`sectionD`	Holds the memory address of an instantiated section within the PEF container; this variable is used by relocation instructions that relocate section addresses. The initial value is the memory address of section 1 if that section is present and instantiated. Otherwise the initial `sectionD` value is `0`. Note that relocation instructions can change the value of `sectionD`; this affects subsequent relocation instructions that refer to this variable. The name `sectionD` is given for convenience only; use of this variable is not restricted to data sections.

Note
The `sectionC` and `sectionD` variables actually contain the memory address of an instantiated section minus the default address for that section. The default address for a section is contained in the `defaultAddress` field of the section header. However, in almost all cases the default address should be `0`, so the simplified definition suffices. ◆

The relocation instructions themselves generally accomplish one of the following functions:

- assign a value to one of the relocation variables
- add an imported symbol's address to the current location (pointed to by `relocAddress`), then increment `importIndex` and `relocAddress`
- add the `sectionC` value to the current location, then increment `relocAddress`
- add the `sectionD` value to the current location, then increment `relocAddress`
- add the `sectionC` value to the current location and increment `relocAddress`, then add the `sectionD` value to the new current location, and increment `relocAddress` again

The Relocation Headers Table

If an instantiated section requires one or more relocations, it has an entry in the relocation headers table. A header entry data structure is of fixed size (12 bytes) and has the form shown in Listing 8-5.

Listing 8-5 Relocation header entry data structure

```
struct PEFLoaderRelocationHeader {
    UInt16  sectionIndex;
    UInt16  reservedA;
    UInt32  relocCount;
    UInt32  firstRelocOffset;
};
```

The header fields are as follows:

- The `sectionIndex` field (2 bytes) designates the section number to which this relocation header refers.
- The `reservedA` field (2 bytes) is currently reserved and must be set to `0`.
- The `relocCount` field (4 bytes) indicates the number of 16-bit relocation blocks for this section.
- The `firstRelocOffset` field (4 bytes) indicates the byte offset from the start of the relocations area to the first relocation instruction for this section.

Note that the `relocCount` field is the number of 16-bit relocation blocks (that is, one half the total number of bytes of relocation instructions). Although most relocation instructions are 16 bits long, some are longer, so the number of complete relocation instructions may be less than the `relocCount` value.

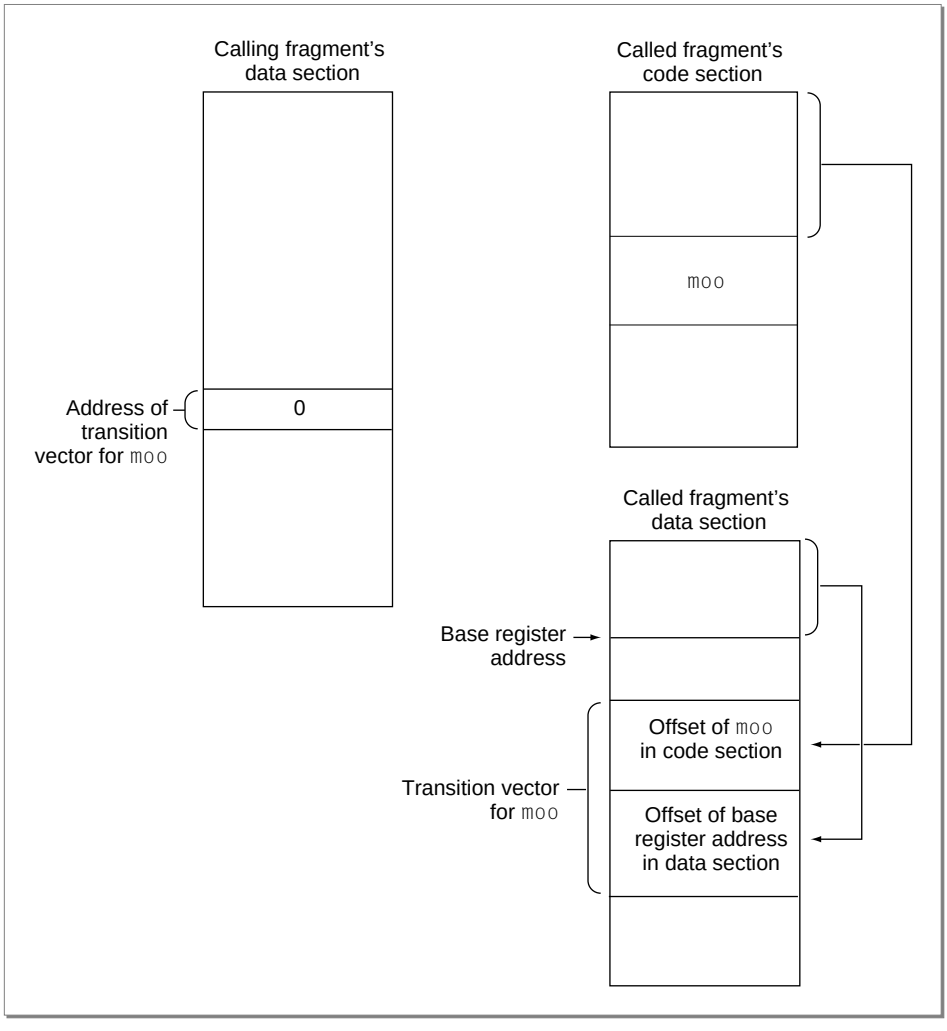
The Relocation Area

The relocation area consists of a sequence of relocation instructions that describe how to fix up pointers to the fragment's own code and data and to imported symbols during the preparation process. These instructions are grouped by section number, and they are accessed through the relocation headers described earlier. See "Relocation Instruction Set" (page 8-27) for a detailed description of the relocation instructions.

A Relocation Example

This section gives an example of how various relocation instructions are used. In this example, a fragment calls the imported routine `moo`. At build time, all pointers to `moo` in the calling fragment are set to `0`, since the compiler or linker cannot know the actual runtime address of the routine. Similarly, in the fragment that contains `moo`, the transition vector for `moo` contains only offset values for the location of its code and its data world. Figure 8-9 shows the unprepared state for the two fragments.

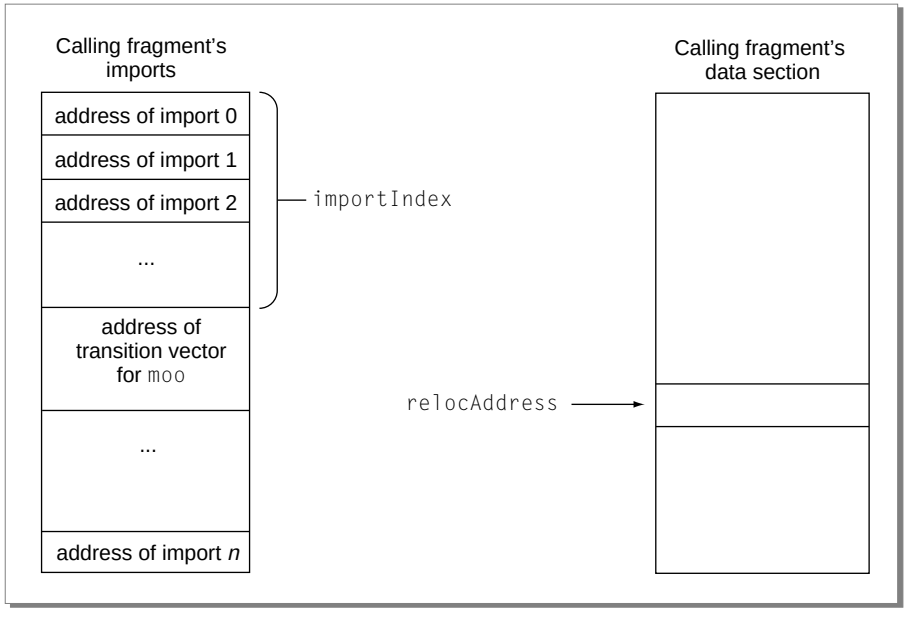
Figure 8-9 Unprepared fragments



After instantiating both fragments, the Code Fragment Manager fixes up the calling fragment's pointer by executing instructions as follows (see Figure 8-10):

1. Set `relocAddress` to point to the data pointer for `moo`.
2. Set `importIndex` to select the imported symbol entry for `moo`.
3. Execute a relocation instruction that adds the address of the imported symbol `moo` (that is, the address of its transition vector) to the 4 bytes at `relocAddress`.

Figure 8-10 Relocations for the calling fragment



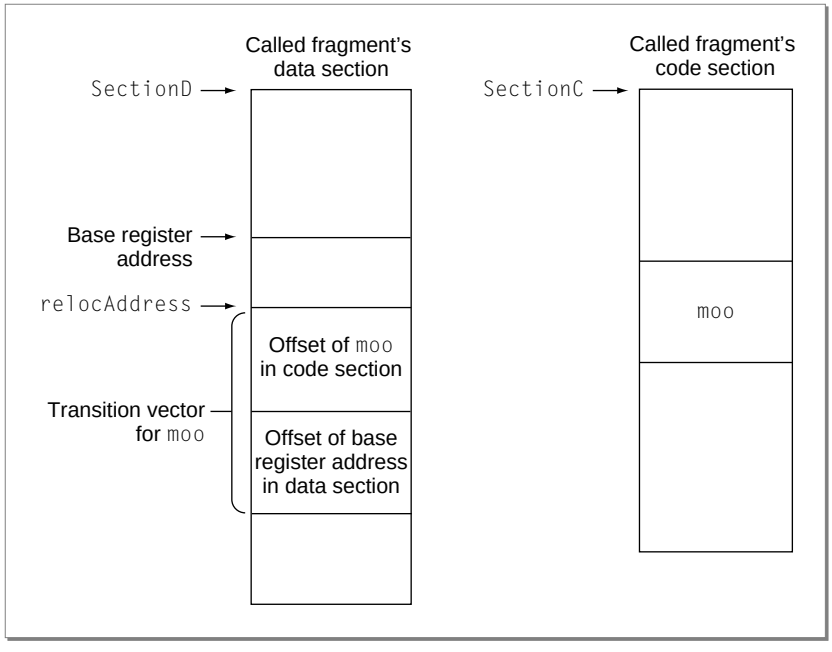
After being fixed up, the calling fragment's pointer now points to the transition vector for `moo`.

The pointers for the called fragment are fixed up as follows (see Figure 8-11):

1. Set `relocAddress` to point to the beginning of the transition vector for `moo`.
2. Set `sectionC` to point to the beginning of the code section containing `moo`.

3. Set `sectionD` to point to the beginning of the called fragment's data section.
4. Execute a relocation instruction that adds `sectionC` to the contents of the location pointed to by `relocAddress`; increments `relocAddress` (4 bytes); adds `sectionD` to the contents of the location pointed to by the new `relocAddress`; and increments `relocAddress` again.

Figure 8-11 Relocations for the called fragment



After being fixed up, the transition vector for `moo` now contains the actual address of `moo` and the base register address for its data world. The routine `moo` is now prepared for execution.

Relocation Instruction Set

Relocation instructions are stored in 2-byte **relocation blocks.** Most instructions take up one block that combines an opcode and related arguments. Instructions that are larger than 2 bytes have an opcode and some of the operands in the

first 2-byte block, with other operands in the following 2-byte blocks. The opcode occupies the upper (higher-order) bits of the block that contains it. Relocation instructions can be decoded from the high-order 7 bits of their first block. Listing 8-6 shows the high-order 7 bits for the currently defined relocation opcode values. Binary values indicated by "x" are "don't care" operands. For example, any combination of the high-order 7 bits that starts with two zero bits (00) indicates the `RelocBySectDWithSkip` instruction.

All currently defined relocation instructions relocate locations as words (that is, 4-byte values).

Listing 8-6 Relocation opcode values

```
enum {

    kPEFRelocBySectDWithSkip    = 0x00,     /* binary: 00xxxxx */

    kPEFRelocBySectC            = 0x20,     /* binary: 0100000 */
    kPEFRelocBySectD            = 0x21,     /* binary: 0100001 */
    kPEFRelocTVector12          = 0x22,     /* binary: 0100010 */
    kPEFRelocTVector8           = 0x23,     /* binary: 0100011 */
    kPEFRelocVTable8            = 0x24,     /* binary: 0100100 */
    kPEFRelocImportRun          = 0x25,     /* binary: 0100101 */

    kPEFRelocSmByImport         = 0x30,     /* binary: 0110000 */
    kPEFRelocSmSetSectC         = 0x31,     /* binary: 0110001 */
    kPEFRelocSmSetSectD         = 0x32,     /* binary: 0110010 */
    kPEFRelocSmBySection        = 0x33,     /* binary: 0110011 */

    kPEFRelocIncrPosition       = 0x40,     /* binary: 1000xxx */
    kPEFRelocSmRepeat           = 0x48,     /* binary: 1001xxx */

    kPEFRelocSetPosition        = 0x50,     /* binary: 101000x */
    kPEFRelocLgByImport         = 0x52,     /* binary: 101001x */
    kPEFRelocLgRepeat           = 0x58,     /* binary: 101100x */
    kPEFRelocLgSetOrBySection   = 0x5A,     /* binary: 101101x */

};
```

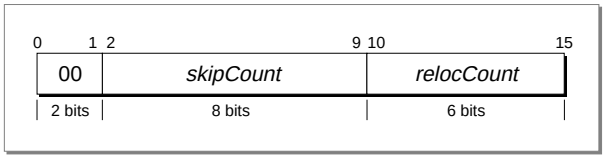
IMPORTANT
If you wish to create your own relocation instructions, the 3 highest order bits must be set (`111xxxx`) to indicate a third-party opcode. All other undocumented opcode values are reserved. ▲

The following sections describe the individual instructions in more detail.

RelocBySectDWithSkip

The `RelocBySectDWithSkip` instruction (opcode 00) has the structure shown in Figure 8-12.

Figure 8-12 Structure of the `RelocBySectDWithSkip` instruction

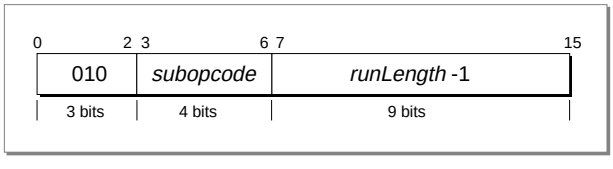


This instruction first increments `relocAddress` by *skipCount* * 4 bytes. It then adds the value of `sectionD` to the next *relocCount* contiguous words. After the instruction is executed, `relocAddress` points just past the last modified word.

The Relocate Value Group

Instructions in the Relocate Value group of opcodes all begin with `010` and have the structure shown in Figure 8-13.

Figure 8-13 Structure of the Relocate Value opcode group



Instructions in this group add a value to the next *runLength* items starting at address `relocAddress`. The subopcode indicates the type and size of the items to be added as shown in Table 8-5. After execution, `relocAddress` points to just past the last modified item.

IMPORTANT
The value stored in this instruction is one less than the actual run length (*runLength*-1). ▲

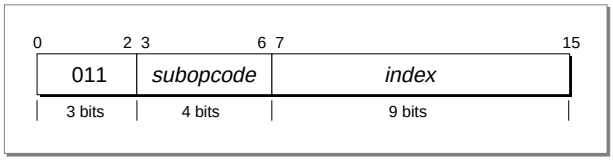
Table 8-5 Subopcodes for the RelocateValue opcode group

Value	Instruction name	Description
`0000`	`RelocBySectC`	Add the value in the variable `sectionC` to the next *runLength* contiguous 4-byte items (words).
`0001`	`RelocBySectD`	Add the value in the variable `sectionD` to the next *runLength* contiguous 4-byte items (words).
`0010`	`RelocTVector12`	Add values to *runLength* 12-byte items as follows: add the value in `sectionC` to the first word and the value in `sectionD` to the second word. No value is added to the third word.
`0011`	`RelocTVector8`	Add values to *runLength* 8-byte items as follows: add the value in `sectionC` to the first word and the value in `sectionD` to the second word.
`0100`	`RelocVTable8`	Add values to *runLength* 8-byte items as follows: add the value in `sectionD` to the first word and do not add any value to the second word.
`0101`	`RelocImportRun`	Add the addresses of a sequence of imported symbols to the next *runLength* contiguous 4-byte items (words). The `importIndex` variable is incremented by 1 after every 4-byte relocation (*runLength* times total).

The Relocate By Index Group

Instructions in the Relocate By Index group all begin with `011` and have the structure shown in Figure 8-14.

Figure 8-14 Structure of the Relocate By Index opcode group



Instructions in this group fix up values according to the subopcode values shown in Table 8-6.

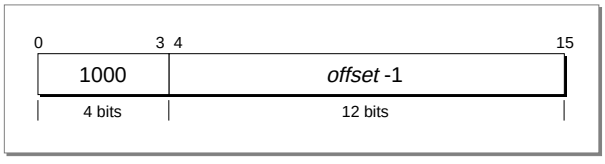
Table 8-6 Subopcodes for the Relocate By Index opcode group

Value	Instruction name	Description
`0000`	`RelocSmByImport`	Add the address of the imported symbol whose index is held in *index* to the word pointed to by `relocAddress`. After the addition, `relocAddress` points to just past the modified word, and `importIndex` is set to *index* +1.
`0001`	`RelocSmSetSectC`	Set the variable `sectionC` to the memory address of the instantiated section specified by *index*.
`0010`	`RelocSmSetSectD`	Set the variable `sectionD` to the memory address of the instantiated section specified by *index*.
`0011`	`RelocSmBySection`	Add the address of the instantiated section specified by *index* to the word pointed to by `relocAddress`. After execution, `relocAddress` points to just past the modified word.

RelocIncrPosition

The `RelocIncrPosition` instruction (opcode 1000) has the structure shown in Figure 8-15.

Figure 8-15 Structure of the `RelocIncrPosition` instruction



This instruction increments `relocAddress` by *offset* bytes. The value of *offset* is treated as an unsigned value.

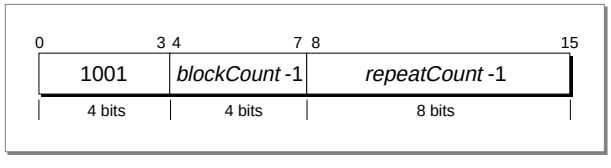
IMPORTANT

The value stored in this instruction is one less than the actual offset (*offset*-1). ▲

RelocSmRepeat

The `RelocSmRepeat` instruction (opcode 1001) has the structure shown in Figure 8-16.

Figure 8-16 Structure of the `RelocSmRepeat` instruction



This instruction repeats the preceding *blockCount* relocation blocks *repeatCount* number of times. Note that you cannot nest this instruction within itself or within the `RelocLgRepeat` instruction.

IMPORTANT

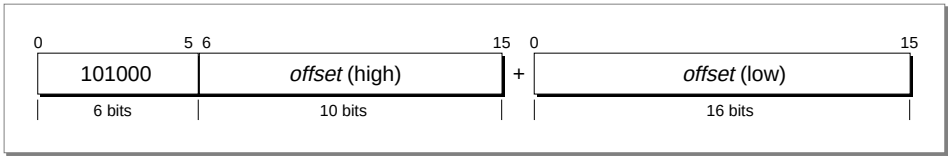
The values of *blockCount* and *repeatCount* stored in this instruction are one less than the actual values. ▲

RelocSetPosition

The `RelocSetPosition` instruction (opcode 101000) takes two relocation blocks (4 bytes) rather than the usual one; the extra bytes allow you to specify an unsigned offset parameter of up to 26 bits.

The `RelocSetPosition` instruction has the structure shown in Figure 8-17.

Figure 8-17 Structure of the `RelocSetPosition` instruction



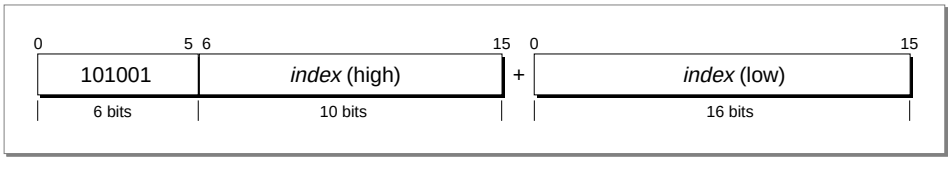
This instruction sets `relocAddress` to the address of the section offset *offset*.

RelocLgByImport

The `RelocLgByImport` instruction (opcode 101001) takes two relocation blocks (4 bytes); the extra bytes allow you to specify an unsigned index parameter of up to 26 bits.

The `RelocLgByImport` instruction has the structure shown in Figure 8-18.

Figure 8-18 Structure of the `RelocLgByImport` instruction

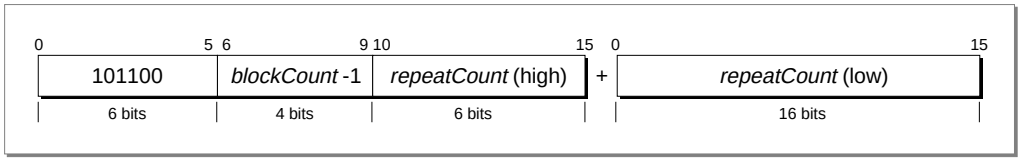


This instruction adds the address of the imported symbol whose index is held in *index* to the word pointed to by `relocAddress`. After the addition, `relocAddress` points to just past the modified word, and `importIndex` is set to *index* +1.

RelocLgRepeat

The `RelocLgRepeat` instruction (opcode 101100) takes two relocation blocks and has the structure shown in Figure 8-19.

Figure 8-19 Structure of the `RelocLgRepeat` instruction



This instruction repeats the preceding *blockCount* relocation blocks *repeatCount* number of times. The `RelocLgRepeat` instruction is very similar to the `relocSmRepeat` (opcode 1001) instruction, but it allows for larger repeat counts.

You cannot nest this instruction, either within itself or within the `relocSmRepeat` instruction.

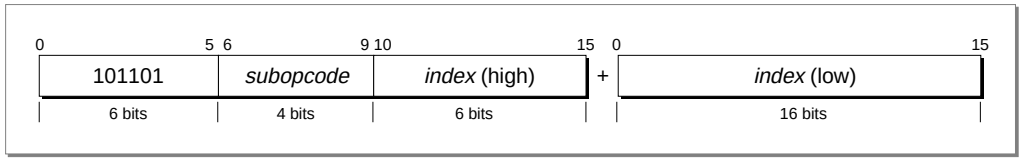
IMPORTANT

Note that the repeat value stored in this instruction is the actual value (*repeatCount*), while for the `relocSmRepeat` instruction the value stored is *repeatCount*-1. The block count value stored is *blockCount*-1 for both repeat instructions. ▲

RelocLgSetOrBySection

The `RelocLgSetOrBySection` instruction (opcode 101101) takes two relocation blocks and has the form shown in Figure 8-20.

Figure 8-20 Structure of the `RelocLgSetOrBySection` instruction



This instruction performs instructions identical to those shown in "The Relocate By Index Group" (page 8-31), but with a larger (up to 22-bit, unsigned) section number. The action specified depends on the value of *subopcode* as shown in Table 8-7.

Table 8-7 Subopcodes for the `RelocLgSetOrBySection` instruction

Subopcode	Action
0000	Add the address of the instantiated section specified by *index* to the word at `relocAddress`. After the addition, `relocAddress` points to just past the modified word. (Same as `RelocSmBySection`.)
0001	Set the variable `sectionC` to the memory address of the instantiated section specified by *index*. (Same as `RelocSmSetSectC`.)
0010	Set the variable `sectionD` to the memory address of the instantiated section specified by *index*. (Same as `RelocSmSetSectD`.)

The Loader String Table

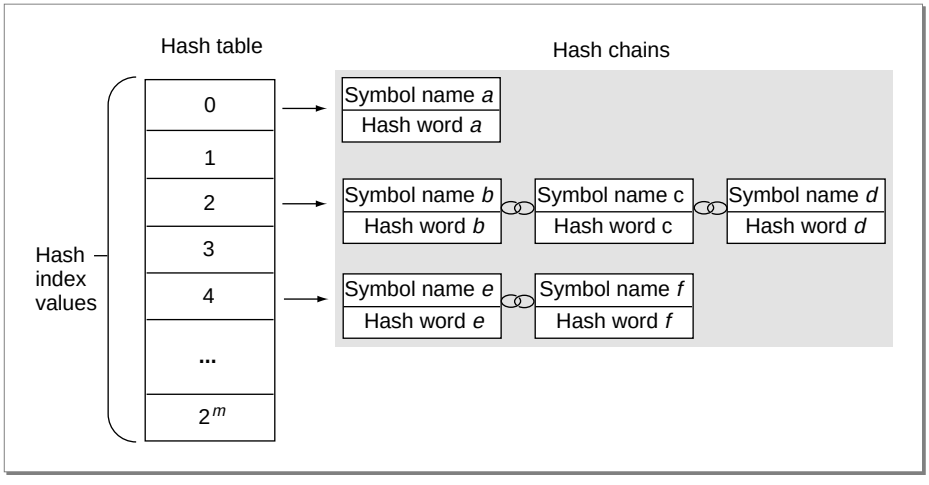
The loader string table contains character strings that specify the names of imported and exported symbols and the names of imported libraries. Strings for imported symbols and imported libraries must be null terminated, but strings referenced by export symbol table entries (that is, strings for exported symbols) do not have this requirement. (The Code Fragment Manager uses the upper 16 bits of the hash value to determine the length of the string). None of the strings contain a Pascal-style length byte.

Exported Symbols

All exported symbols in a PEF container are stored in a hashed form, allowing the Code Fragment Manager to search for them efficiently when preparing a fragment. **Hashing** is a method of processing and organizing symbols so they can be searched for quickly.

PEF uses a modified version of the traditional hash table. The traditional model is shown in Figure 8-21.

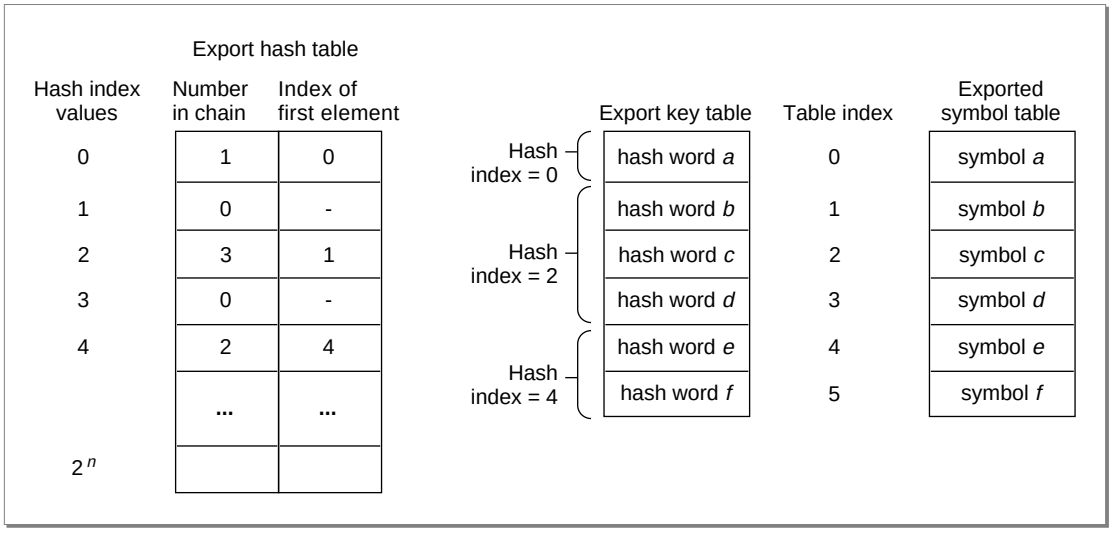
Figure 8-21 A traditional hash table



A hash word is computed for every symbol and a hash index value is computed for every hash word. The hash words are grouped together in hash chains according to their index values, and each chain corresponds to an entry in the hash table.

The PEF implementation, as shown in Figure 8-22, effectively flattens the traditional hash table. Functionally the hash tables in Figure 8-21 and Figure 8-22 are identical.

Figure 8-22 Flattened hash table implementation



Each hash chain is stored consecutively in the export key table and the exported symbol table. For each hash index value, the hash table stores the number of entries in its chain and the starting table index value for that chain.

The general procedure for creating a hashed data structure is as follows:

1. Compute the number of hash index values. This value is based on the number of exported symbols in the container. See "The Exported Symbol Count to Hash Table Size Function" (page 8-42) for a suggested method of calculating this value.
2. Compute the hash word value and hash index value for every exported symbol. (The hash index value is dependent on both the symbol and the size of the hash table.) See "The Name to Hash Word Function" (page 8-41) and "The Hash Word to Hash Index Function" (page 8-42) for details of the required calculations.
3. Sort the exported symbols by hash index value. This procedure effectively indexes the exported symbols. Each symbol has a table index value that references its hash word in the export key table and an entry in the exported symbol table.

4. Construct the hash table using the size determined in step 1. Each hash table entry contains a chain count indicating the number of exported symbols in the chain (that is, the number that have this hash index value) and the offset in the export key and symbol tables to the first symbol in the chain.

The Code Fragment Manager can search for exported symbols by name or by table index number. When searching for a symbol by (zero-based) table index number, the Code Fragment Manager looks up the index value in the exported symbol table to obtain a pointer to the name of the symbol. Then it uses the same index to get the hash word value of the symbol in the export key table. (The length of the name is encoded in the hash word.)

Searching for exported symbols by name is somewhat more complicated. The Code Fragment Manager first computes the hash word of the symbol it is trying to locate. Then it computes a hash index value from the hash word and the size of the hash table. Using this value as an index into the hash table, the Code Fragment Manager obtains a chain count value and a table index value for the first entry in the hash chain (as determined in step 4). Then, beginning at the table index value, it searches the export key table for a hash word to match the one it previously calculated. If the Code Fragment Manager finds a match, it uses the matching table index value to look up the name in the symbol table. If the symbol names match, the Code Fragment Manager returns information about the symbol. If the Code Fragment Manager cannot find a match after searching the number of entries equivalent to the chain count value, it marks the symbol as not found.

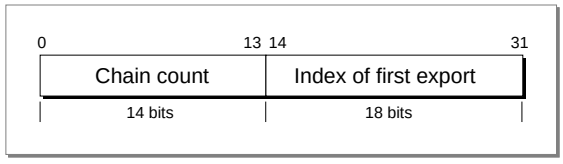
The sections that follow describe the elements of the hashed data structure in more detail.

The Export Hash Table

The number of entries in the hash table is 2 raised to the value in the `exportHashTablePower` field of the loader header (page 8-18). The number of entries is determined from the number of exported symbols. If there are no exports, the table still contains one entry. See "Hashing Functions" (page 8-41) for details of the hashing process and the suggested method for computing the number of hash table entries.

A hash table entry is of fixed size (4 bytes) and has the form shown in Figure 8-23.

Figure 8-23 A hash table entry



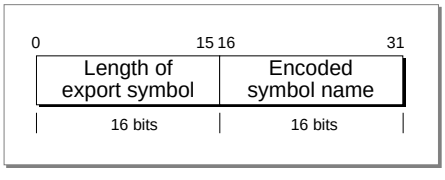
The field values are as follows:

- The first field (14 bits) contains the number of items in this chain.
- The second field (18 bits) contains the table index value of the first symbol in the chain (see Figure 8-22 (page 8-37)).

The Export Key Table

The export key table contains a key (a hash word) for every exported symbol. The structure of a hash word is fixed (4 bytes) and has the form shown in Figure 8-24.

Figure 8-24 A hash word



- The first field contains the length of the export symbol name in bytes.
- The second field contains the name of the symbol encoded using a hash key.

For more information about calculating the hash word, see "The Name to Hash Word Function" (page 8-41).

The Exported Symbol Table

The exported symbol table contains an entry for every symbol exported by the fragment. All exports with a given hash index value are grouped together in the symbol table (see Figure 8-22 (page 8-37)).

An exported symbol table entry data structure is of fixed size (10 bytes) and has the form shown in Listing 8-7.

Listing 8-7 Exported symbol table entry data structure

```
struct PEFExportedSymbol {
    UInt32  classAndName;
    UInt32  symbolValue;
    SInt16  sectionIndex;
};
```

Note
Each entry is 10 bytes long. No padding bytes are used between successive entries. ◆

The fields in the entry are as follows:

- The `classAndName` field (4 bytes) contains two entries:
  - The first byte designates the symbol class of the exported symbol. See Table 8-3 (page 8-21) for a listing of classes. Flag bits for exported symbols are reserved for future use.
  - The following 3 bytes designate the offset from the beginning of the loader string table to the name of the symbol. The name of the symbol is not null terminated, but you can determine the length of the string from the upper 2 bytes of the symbol's hash word (found in the export key table).
- The `symbolValue` field (4 bytes) typically indicates the offset from the beginning of the symbol's section to the exported symbol.
- The `sectionIndex` field (2 bytes) indicates the number of the section that contains this symbol. Note that this is a signed field.

The `symbolValue` field has special meaning when the section number is negative. If the section number is `-2`, the `symbolValue` field contains an absolute address. If the section number is `-3`, the `symbolValue` field contains an imported symbol index, indicating that the imported symbol is being reexported.

Hashing Functions

This section describes hashing algorithms used to create the hashed data structure for exported symbols.

The Name to Hash Word Function

The hash word function computes a 32-bit hash word for a symbol name. The upper 16 bits contains the length of the name, and the symbol name is encoded using a hash key in the lower 16 bits. You are required to use this algorithm to calculate the hash word. Listing 8-8 shows a C implementation of the hash word function.

Listing 8-8 Hash word function

```
/* Computes a hash word for a given string. nameText points to the */
/* first character of the string (not the Pascal length byte). The */
/* string may be null terminated. */

enum {
    kPEFHashLengthShift     = 16,
    kPEFHashValueMask       = 0x0000FFFF
};

UInt32 PEFComputeHashWord (BytePtr nameText, UInt32 nameLength)
{
    BytePtr charPtr     = nameText;
    SInt32  hashValue   = 0;
    UInt32  length      = 0;
    UInt32  limit;
    UInt32  result;
    UInt8   currChar;

    #define PseudoRotate(x)  ( ( (x) << 1 ) - ( (x) >> 16 ))

    for (limit = nameLength; limit > 0; limit -= 1)
    {
        currChar = *charPtr++;
        if (currChar == NULL) break;
        length += 1;
```

```
        hashValue = PseudoRotate (hashValue) ^ currChar;
    }

    result = (length << kPEFHashLengthShift) |
    ((UInt16) ((hashValue ^ (hashValue >> 16)) & kPEFHashValueMask));

    return result;

} /* PEFComputeHashWord () */
```

The Hash Word to Hash Index Function

The hash index (or hash slot number) function converts the 32-bit hash word value into a small index number.You are required to use this algorithm for calculating the index number. Listing 8-9 shows the hash word to hash index function.

Listing 8-9 Hash word to hash index function

```
#define PEFHashTableIndex(fullHashWord,hashTablePower) \
    ( ( (fullHashWord) ^ ((fullHashWord) >> (hashTablePower)) ) & \
    ((1 << (hashTablePower)) - 1) )
```

The Exported Symbol Count to Hash Table Size Function

Listing 8-10 shows a suggested method of calculating the hash table size. (This algorithm provides a good tradeoff between minimizing search time and minimizing table size, but you may substitute a similar algorithm.) The hash table size function computes the size of the hash table based on the number of exported symbols in the PEF container. The number of hash table entries is always a power of 2. The function in Listing 8-10 returns the value of the exponent. The value `kExponentLimit` can be arbitrary, but it must not exceed 30. The constant `kAverageChainLimit` is normally set to `10`, but you can adjust this to make a trade off between the size of the chain and search time.

Listing 8-10 Exported symbol count to hash table size function

```
UInt8   PEFComputeHashTableExponent (SInt32 exportCount)
{
    SInt32  exponent;

    const SInt32    kExponentLimit      = 16;
    const SInt32    kAverageChainLimit  = 10;

    for (exponent = 0; exponent < kExponentLimit; exponent += 1) {
        if ((exportCount / (1 << exponent)) < kAverageChainLimit)
            break;
    }

    return exponent;

}   /* PEFComputeHashTableExponent () */
```

PEF Size Limits

The PEF structure has the following size limits:

- The total size of the container cannot be larger than 4 GB.
- The maximum offset allowed into the section name table is 2 GB.
- The total number of sections cannot exceed 65,535.
- The total number of instantiated sections (that is, those containing code or data) cannot exceed 32,767.
- The maximum size of the loader string table is 16 MB.
- The total number of imported symbols is limited to 2^{26}. However, the number of reexported imports is limited to 2^{24}.
- The number of exported symbols is limited to 2^{18}.
- A single hash chain cannot contain more than 16,384 entries.

In general, 32-bit integers (`UInt32`) are used to store size and count values in PEF containers, resulting in a maximum allowable integer of 2^{32}. In many cases, this is a theoretical rather than actual limit, since other PEF limitations may restrict the largest allowable value.

Note that the Code Fragment Manager itself imposes limits that are not related to the PEF specification. For example, there is a length limit of 255 characters for imported and exported symbol names and a 63 character limit for imported library names. For specifics, check the current Code Fragment Manager documentation.

CFM-68K Application and Shared Library Structure

Contents

This chapter describes the file structure of CFM-68K runtime applications and shared libraries. You need to read this section only if you need specific details about segment structure and the storage of items such as transition vectors and jump table entries in CFM-68K runtime programs.

Note
Some sections specifically describe MPW implementations for the CFM-68K runtime environment. Other implementations are possible, however. ◆

CFM-68K Application Structure

Although CFM-68K runtime shared libraries are virtually identical to their PowerPC counterparts, CFM-68K runtime applications are hybrids that retain the segmented form of classic 68K applications.

CFM-68K runtime applications use some classic 68K structures (`'CODE'` resources, for example), but many of these structures have been modified for the CFM-based architecture. CFM-68K applications have different segment headers and jump tables, as well as a new table for transition vectors. The `%A5Init` segment does not exist in CFM-68K applications, and the `'CODE'0` resource does not hold the jump table. The following sections describe the CFM-68K application structure in detail.

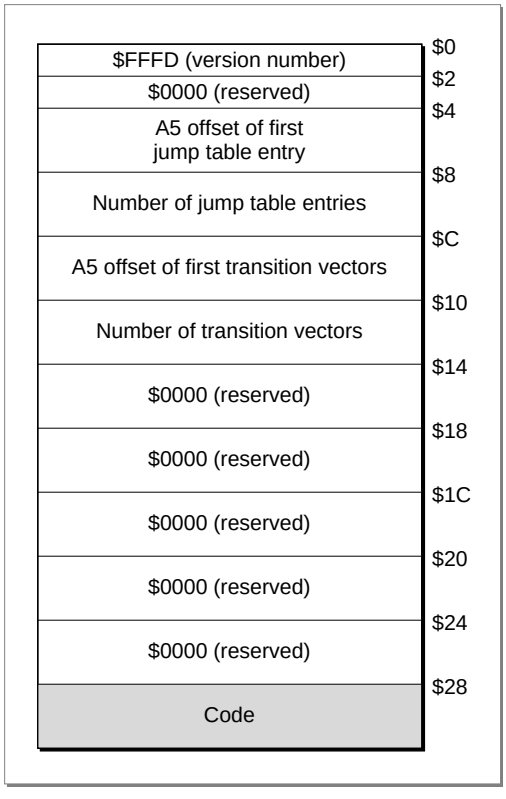
Note
If you are not familiar with the structure of classic 68K applications, you may want to refer to Chapter 10, "Classic 68K Runtime Architecture," as you read this section. ◆

The Segment Header

Each CFM-68K runtime segment contains a header that gives information about the segment. Figure 9-1 shows the structure of a CFM-68K runtime segment header.

Note
The CFM-68K runtime segment header is the same size as a classic 68K far model (32-bit everything) header (see Figure 10-11 (page 10-24)), but it contains different information. ◆

Figure 9-1 Structure of a CFM-68K runtime segment header



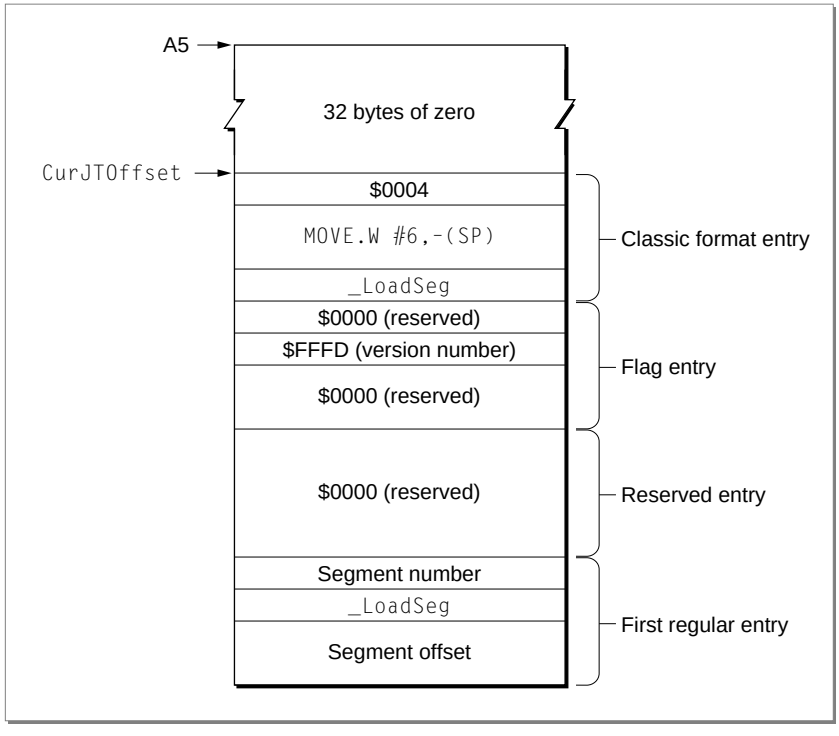
The version number $FFFD indicates that the segment header was built for the CFM-68K runtime architecture. This value must match the version number in the jump table flag entry (see Figure 9-2 (page 9-5)).

Note
In MPW you must build your application with the same size constraints as a classic 68K near model program unless you specify the `-bigseg` compiler option. ◆

The Jump Table

Figure 9-2 shows the structure of a CFM-68K runtime jump table. This jump table is similar to the classic 68K far model jump table as shown in Figure 10-10 (page 10-23).

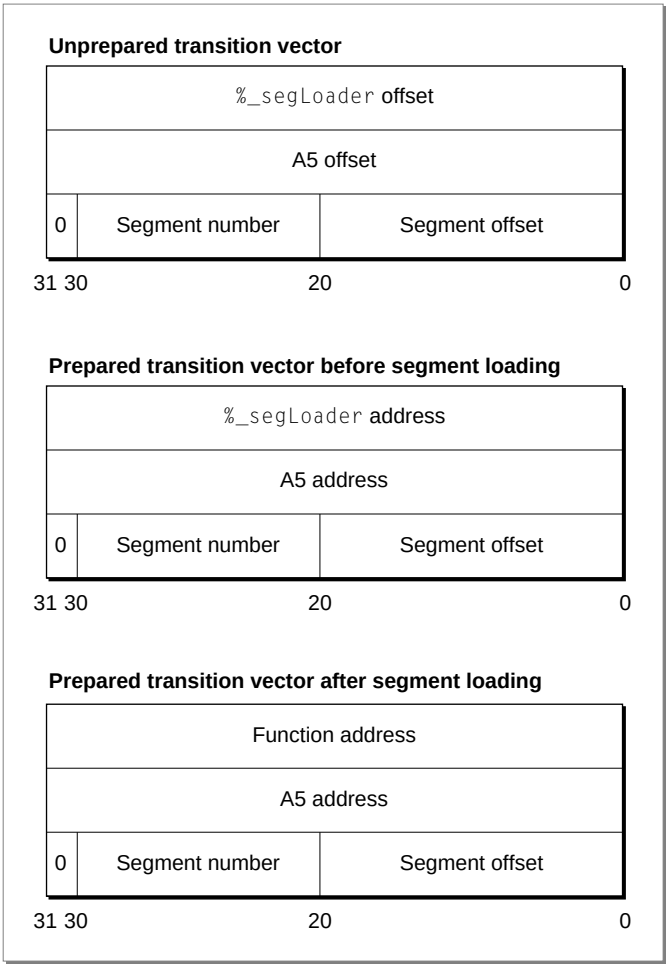
Figure 9-2 CFM-68K runtime jump table structure



Transition Vectors and the Transition Vector Table

The transition vector table resides in the direct data world (the A5 world in classic 68K) above the jump table. It contains a transition vector for every exported routine and every routine whose address is accessed in any way. The transition vectors contain the entry point address for the desired routine and the value to be placed in the A5 base register when the routine executes. Figure 9-3 shows the structure of an application transition vector.

Figure 9-3 An application transition vector



The Code Fragment Manager sets the `%_segLoader` address and A5 address portions of the transition vector at preparation time. (See “The 'rseg' 1 Resource” (page 9-10) for more information about the `%_segLoader` routine.) The application transition vector is larger than the corresponding shared library transition vector (12 bytes versus 8 bytes) because it needs additional segment information to properly address routines in a segmented application.

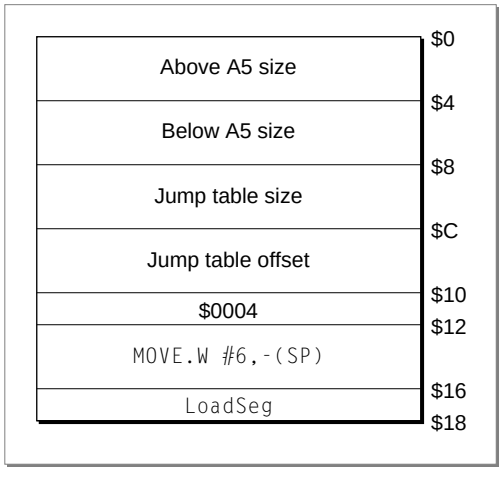
The segment offset field in a transition vector contains a word (2 byte) offset. This differs from a jump table entry’s segment offset field, which contains a byte offset.

The 'CODE' 0 Resource

A CFM-68K runtime application’s `'CODE'0` resource contains a small “start-up” jump table that loads and executes the code that launches the application. In MPW, this code is stored in the `'CODE'6` resource. Figure 9-4 shows the structure of a CFM-68K runtime `'CODE'0` resource.

Note
In classic 68K applications, the `'CODE'0` resource contains the application’s jump table. ◆

Figure 9-4 The `'CODE'0` resource



The 'CODE' 6 Resource

In MPW, code stored in the `'CODE'6` resource handles the launching of CFM-68K runtime applications for the Process Manager in System 7.1.

Note
The Process Manager in System 7.5 or later can launch CFM-68K runtime applications directly without having to execute routines in the `'CODE'6` resource. ◆

The `CFM Launch` code segment in the `'CODE'6` resource takes the following steps when launching a CFM-68K runtime application.

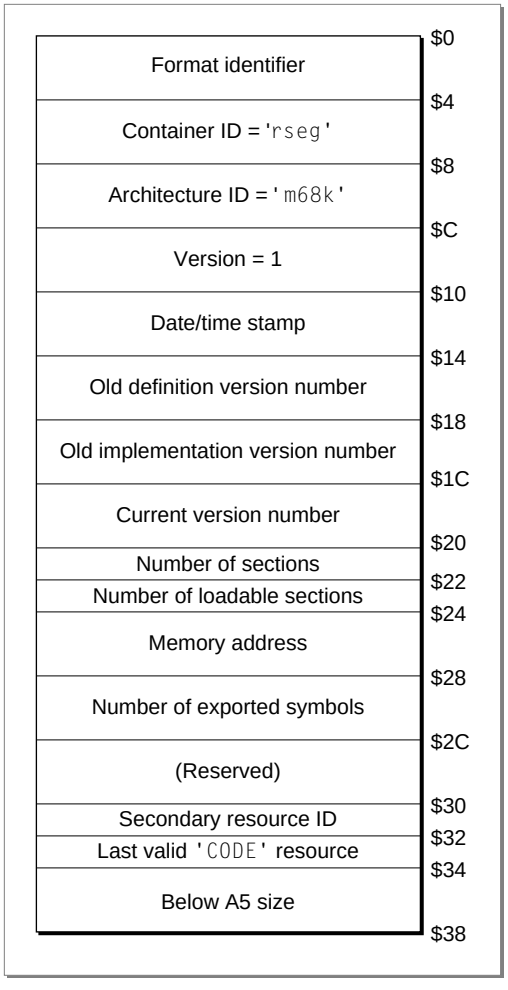
1. Checks to see if the computer is a PowerPC-based machine. If so, the `CFM Launch` segment displays the message, "Sorry, this application doesn't run on PowerPC platforms. You may only run it on 68K platforms." If you want to create a custom version of this message, you must install a `'STR '` resource with ID -20227 in your CFM-68K runtime application.
2. Checks to see that the CFM-68K Runtime Enabler is installed on the computer. If the CFM-68K Runtime Enabler is missing, the `CFM Launch` segment displays the message, "This application requires installation of the CFM-68K Runtime Enabler." If you want to create a custom version of this message, you must install a `'STR '` resource with ID -20029 in your CFM-68K runtime application.
3. Reads the `'cfrg'0` resource and calls the Code Fragment Manager to select the proper fragment from the `'cfrg'0` entries.
4. Tells the Code Fragment Manager to prepare the application fragment, along with any necessary import libraries.
5. Adds code to the `ExitToShell` routine to perform the necessary CFM clean-up operations when an application quits or aborts.
6. Calls the application's main entry point.

The 'rseg' 0 Resource

The `'rseg'0` resource is the resource loaded and retained by the Code Fragment Manager and is the fragment referenced from the `'cfrg'0` resource. Since the Code Fragment Manager does not release an "active" fragment, the `'rseg'0` resource does not contain the executable fragment, but only a small data structure. This structure specifies the location of the actual executable fragment as well as some additional information about the fragment. The actual

executable fragment is stored in the 'rseg'1 resource, which can be released after the application launch procedures are completed. The 'rseg'0 resource contains a copy of the PEF container header from the 'rseg'1 resource along with other information, as shown in Figure 9-5. For more information about PEF headers, see "The Container Header," beginning on page 8-4.

Figure 9-5 The 'rseg'0 resource



The 'rseg' 1 Resource

The `'rseg'1` resource holds a PEF container consisting of the following sections:

- A data section containing the application's jump table, transition vector table, and global data, all in a compressed format. This section replaces the `%A5Init` segment used for classic 68K runtime applications.
- A loader section that specifies the import libraries needed by the application. This section also includes a list of symbols imported from each library and a list of symbols (if any) exported from the application.
- A code section containing the `%_segLoader` routine. This code patches the `_LoadSeg` and `_UnloadSeg` A-line instructions, so they can function properly in the CFM-68K runtime environment.

See Chapter 8, "PEF Structure," for more information about PEF containers.

The Code Fragment Manager uses the `'rseg'` resources to create a direct data area, perform A5 relocations, and bind shared libraries to the application. While preparing the launch of a CFM-68K application, the Code Fragment Manager stores the `'rseg'1` resource in the application heap (much the way the `%A5Init` segment is stored for classic 68K applications). After preparations are complete, the Code Fragment Manager releases the `'rseg'1` resource.

CFM-68K Shared Library Structure

In some development environments, creating a CFM-68K shared library involves first creating a segmented version of the library and then flattening it to produce a contiguous program that is stored in the file's data fork. In MPW, the mechanism for flattening segmented shared libraries is the MakeFlat tool. This section describes what conversions are necessary to go from a segmented state to a flattened state and how MakeFlat implements these conversions.

You need to read this section in either of these two cases:

- You want to understand how the MPW MakeFlat tool flattens CFM-68K shared libraries.
- You are writing a library flattening tool and want to understand what conversions are necessary.

An unflattened shared library has a structure very similar to that of a CFM-68K runtime application. The main differences are as follows:

- The transition vectors are 8 bytes long instead of 12.
- The PEF container's data section is not compressed.
- The `'cfrg'0` resource indicates that the fragment is a library, not an application.

The structure changes radically, however, when you flatten the segmented library using the MakeFlat tool. MakeFlat makes the following changes to a segmented shared library:

- Converts the shared library's `'CODE'` resources (except for `'CODE'0` and `'CODE'6`) into code sections in the output PEF container.
- Modifies the PEF relocations.
- Converts jump table entries and transition vectors to their flattened state.
- Compresses the PEF container's data section.
- Creates a new `'cfrg'0` resource specifying the new location of the PEF container.
- Adds a debug section to the output PEF container so you can use the 68K Macintosh Debugger to debug shared libraries.
- Adds code to properly call static constructor or destructor routines if they exist in the shared library.

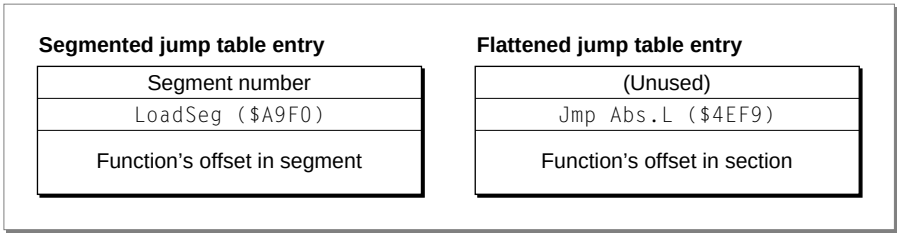
After making these changes, MakeFlat writes the PEF container to the data fork of the output file.

The following sections describe some of the conversions in greater detail.

Jump Table Conversion

When MakeFlat flattens jump table entries, it changes the addressing method from one that is segment oriented to one that is code section oriented. This involves removing the segment number (since it serves no purpose in a flat file), changing the `LoadSeg` instruction to a `Jmp Abs.L` instruction, and copying the routine's offset into the new entry. Then, MakeFlat generates a relocation instruction for each jump table entry that adds the code section's address to the routine's offset. Figure 9-6 compares the two jump table versions.

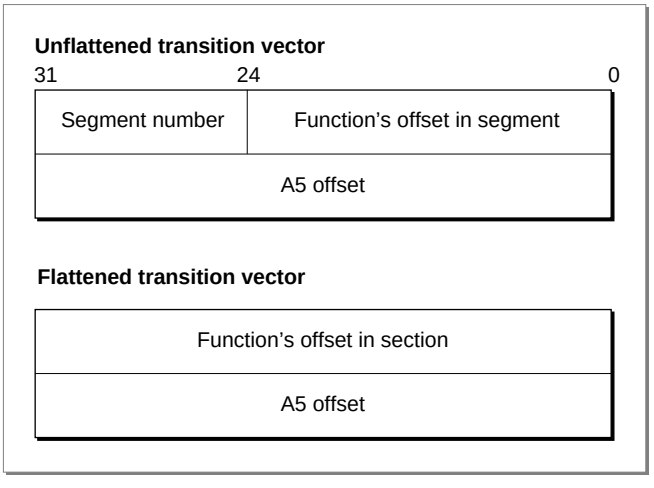
Figure 9-6 Segmented versus flattened jump table entries



Transition Vector Conversion

As with the jump table entries, MakeFlat converts the transition vector addressing scheme from one that is segment oriented to one that is code section oriented. MakeFlat generates a relocation instruction for each transition vector that adds the section's address and A5 address to the offset in each transition vector. Figure 9-7 shows a transition vector before and after conversion (flattening).

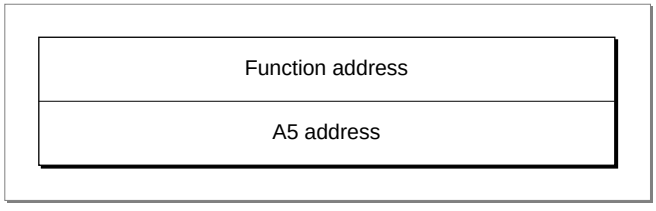
Figure 9-7 A transition vector before and after flattening



Note that the function offset in the unflattened transition vector is a word offset, and the A5 offset is a byte offset. In a flattened transition vector, MakeFlat has converted the function's word offset into a byte offset.

At runtime, the transition vector offset values are replaced with absolute addresses, as shown in Figure 9-8.

Figure 9-8 A transition vector at runtime



Static Constructors and Destructors

In MPW, if the input shared library contains static constructors or destructors, the MakeFlat tool performs special processing to ensure these routines are called at the proper time.

MakeFlat adds a block of data to the top of the A5 world and adds a new code section. The data block consists of two new transition vectors, offsets to the library's original initialization and termination routines, and the contents of the code segment `%_Static_Constructor_Destructor_Pointers`. The new code section, `%_CPlus_Static_Init_Term`, contains the two routines that call the static constructors and destructors. These two routines are then marked as being the library's initialization and termination routines.

Note
Because MakeFlat takes care of static object construction and destruction, you do not need to call the MPW routines `__init_lib` and `__term_lib` when creating your own initialization and termination routines for shared libraries. However, when creating routines for CFM-68K runtime applications, you must call the corresponding `__init_app` and `__term_app` routines. ◆

Classic 68K Runtime Architecture

Contents

The classic 68K runtime architecture is the original Macintosh runtime architecture, designed for computers running a Motorola 68000-series microprocessor. Applications are stored as segments that can be loaded into the application heap as necessary. The application space contains the application heap, the application stack, and the A5 world.

This chapter gives an overview of the classic 68K runtime architecture, with information about the following topics:

- the A5 world
- program segmentation
- the jump table
- addressing limitations of the original classic 68K architecture and MPW solutions for bypassing these limitations

The first three sections, which discuss the A5 world, program segmentation, and the jump table, assume the **near model** classic 68K architecture. Programs built using the near model rely on 16-bit addressing for code and data. The sections that follow introduce the **far model,** which relies on 32-bit addressing for code and data. Note that you have the option of incorporating only some of the far model characteristics when building your application.

For additional information you should consult the various volumes of the *Inside Macintosh* series.

Note
Classic 68K runtime code cannot use shared libraries. However, classic 68K runtime code can run transparently under emulation on PowerPC-based computers. ◆

The A5 World

Every classic 68K application contains an **A5 world,** an area of memory that stores the following items:

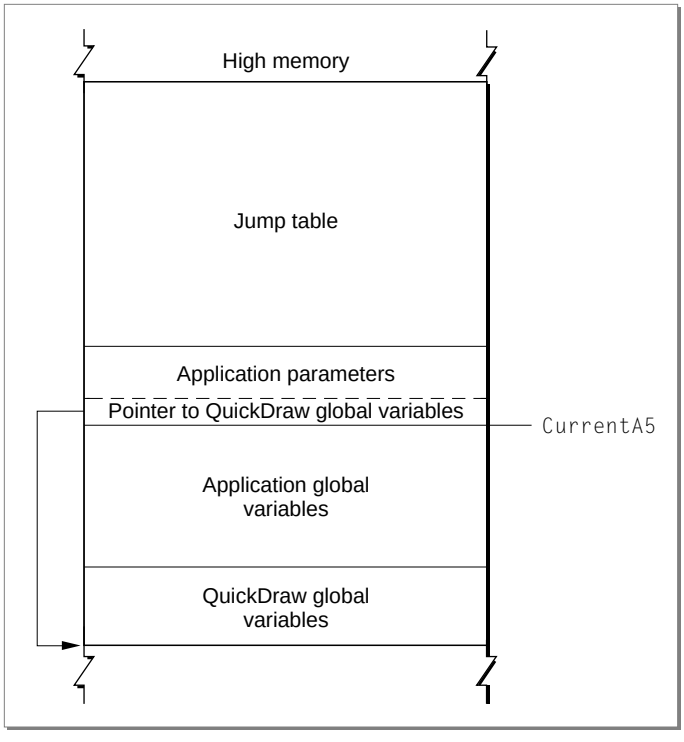
- the jump table, which allows the application to make calls between segments
- the application's global variables

- the application's QuickDraw global variables, which contain information about the drawing environment
- the application parameters, which are reserved for use by the Mac OS

The data is referenced as offsets from the value of the A5 register, hence the name A5 world. The application's global variables and QuickDraw global variables are referenced with negative offsets from A5, while application parameters and jump table entries are referenced with positive offsets.

Figure 10-1 shows a classic 68K A5 world.

Figure 10-1 Classic 68K A5 world



The system global variable `CurrentA5` holds the value of the A5 register.

Program Segmentation

The classic 68K runtime architecture reflects the need for maximum memory efficiency in the original Macintosh computer which had 128 KB of RAM and an MC68000 CPU. To run large applications in this limited memory environment, Macintosh applications were broken up into **segments** (`'CODE'` resources) that could be loaded into the application heap as necessary.

When you compile and link a program, the linker places your program's routines into code segments and constructs `'CODE'` resources for each program segment. The Process Manager loads some code segments into memory when you launch an application. Later, if the code calls a routine stored in an unloaded segment, the Segment Manager loads that new segment into memory. These operations occur automatically by using information stored in the application's jump table and in the individual code segments themselves.

Note that although the Segment Manager loads segments automatically, it does not unload segments. The Segment Manager locks the segment when it is first loaded into memory and any time thereafter when routines in that segment are executing. This locking prevents the segment from being moved during heap compaction and from being purged during heap purging.

Your development environment lets you specify compiler directives to indicate which routines should be grouped together in the same segment. For example, if you have code that is not executed very often (for example, code for printing a document), you can store that in a separate segment, so it does not occupy memory when it is not needed. Here are some general guidelines for grouping routines into segments:

- Group related routines in the same segment.
- Put your main event loop into the **main segment** (that is, the segment that contains the main entry point).
- Put any routines that handle low-memory conditions into a locked segment (usually the main segment). For example, if your application provides a grow-zone function, you should put that function in a locked segment.
- Put any routines that execute at interrupt time, including VBL tasks and Time Manager tasks, into a locked segment.
- Any initialization routines that are executed only once at application startup time should be put in a separate segment. This grouping allows you to

unload the segment after executing the routines. However, routines that allocate non relocatable objects (for example, `MoreMasters` or `InitWindows`) in your application heap should be called in the main segment, before loading any code segments that will later be unloaded. If you put such allocation routines in a segment that is later unloaded and purged, you increase heap fragmentation.

A typical strategy is to unload all segments except segment 1 (the main segment) and any other essential code segments each time through your application's main loop.

To unload a segment you must call the `UnloadSeg` routine from your application. The `UnloadSeg` routine does not actually remove a segment from memory, but merely unlocks it, indicating to the Segment Manager that it may be relocated or purged if necessary. To unload a particular segment, you pass the address of any externally referenced routine contained in that segment. For example, if you wanted to unload the segment that contains the routine `happyMoo`, you can execute the following:

```
UnloadSeg(&happyMoo);
```

▲ **WARNING**
Before you unload a segment, make sure that your application no longer needs it. Never unload a segment that contains a completion routine or other interrupt task (such as a Time Manager or VBL task) that might be executed after the segment is unloaded. Also, you must never unload a segment that contains routines in the current call chain. ▲

The Jump Table

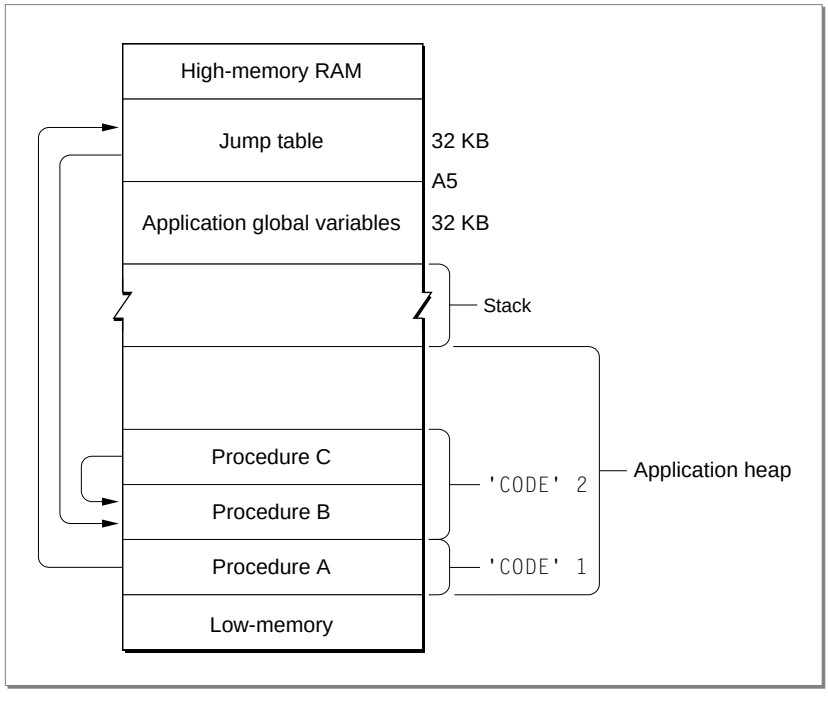
The loading and unloading of segments are controlled by the linker and the Segment Manager through the use of the **jump table** (`'CODE'0`), a data structure created by the linker. The jump table is always located at a fixed offset above A5 as shown previously in Figure 10-1.

The jump table is used to track the state (loaded or unloaded) and the location of `'CODE'` resources. The jump table keeps track of the location of each `'CODE'` resource and the offset of each routine inside each segment.

- If one routine needs to call another routine in a different segment **(intersegment reference),** it must go through the jump table to determine the address where the other routine starts. If the segment containing the externally referenced routine is unloaded, it must be loaded before jumping to the routine address.
- If a routine calls another routine in its own segment **(intrasegment reference),** it does not need the jump table. Although `'CODE'` resources move in the heap, their contents are constant, so the routines always keep a constant distance apart and can be accessed using a self-relative (that is, a PC-relative) branch.

Figure 10-2 shows a call that goes through the jump table and a call that uses self-relative branching.

Figure 10-2 Using the jump table and using self-relative branching



When procedure A calls procedure B, procedure A must go through the jump table because the procedures are in different segments. But procedure C can call procedure B without going through the jump table because the procedures are in the same segment.

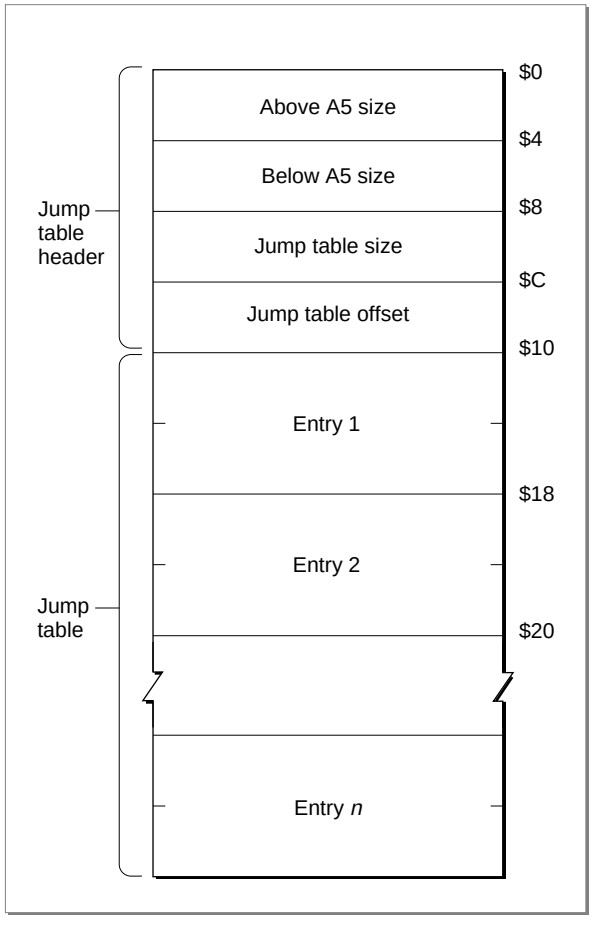
If you trace through code and see an instruction such as

```
JSR     60(A5)
```

you are looking at a call to a routine in another code segment—that is, a call that must go through the jump table. Remember that A5 is used to reference the application's global variables and the jump table. Negative offsets from A5 reference global variables, while positive offsets that are greater than 32 refer to jump-table entries.

The jump table is created by the linker when you build your application, and it is stored in the `'CODE'0` resource (sometimes called *segment 0*). The structure of the `'CODE'0` resource is shown in Figure 10-3.

Figure 10-3 The 'CODE'0 resource



The elements of the 'CODE'0 resource are as follows:

- Above A5 size. The size (in bytes) from the location pointed to by A5 to the upper end of the application space.
- Below A5 size. The size (in bytes) of the application's global variables plus the QuickDraw global variables.
- Jump table size. The size of the jump table. The jump table contains one 8-byte entry for each externally referenced routine.

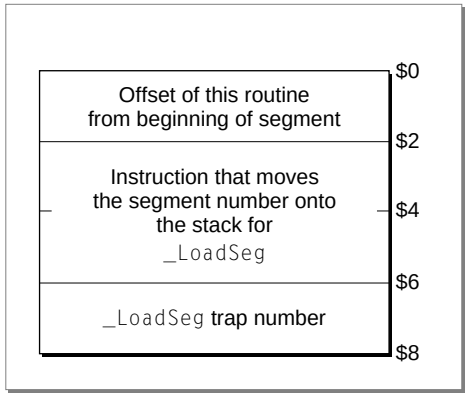
- Jump table offset. The offset (in bytes) of the jump table from the location pointed to by A5. This offset is stored in the global variable `CurJTOffset`.
- Jump table. A contiguous list of jump table entries.

Note
For all applications, the offset to the jump table from the location pointed to by A5 is 32. The number of bytes above A5 is 32 plus the length of the jump table. ◆

When the application is launched, the Segment Manager uses this information to place the jump table in the A5 world.

The linker creates a jump table entry for every routine that is called by a routine from a different segment. All entries for a particular segment are stored contiguously in the jump table. The structure of the entry varies depending on whether the referenced routine is in a loaded or unloaded segment. If the segment has not been loaded into memory, the jump table entry has the structure shown in Figure 10-4.

Figure 10-4 An unloaded jump table entry



Note
The jump table structure for unloaded segments is different if you are building with the `-model far` option. See "The Far Model Jump Table" (page 10-20) for more details. ◆

A call that goes through the jump table has the form

`JSR` *offset* `(A5)`

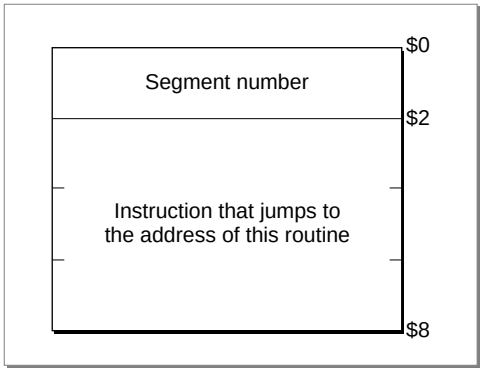
where *offset* is the offset of the jump table entry for the routine from A5 plus 2 bytes. This results in the execution of the `MOVE.W #`*n*`, -SP` instruction, which places the number of the segment containing the routine on the stack. (The jump table refers to segments by the segment numbers assigned by the linker.)

The next instruction invokes the `_LoadSeg` trap, which loads the specified segment into memory. Then the Segment Manager can transform all the jump table entries for the segment into their loaded states as follows:

1. The Segment Manager loads the segment, locks it, double-dereferences it, and adds the offset, which is stored in the first word of the unloaded entry. This results in the actual address of the routine.
2. The Segment Manager then builds the loaded entry format: it stores the segment number in the entry's first 2 bytes, and it stores a `JMP` instruction to the address it has calculated in the entry's last 6 bytes.

Figure 10-5 shows the structure of a loaded jump table entry.

Figure 10-5 A loaded jump table entry



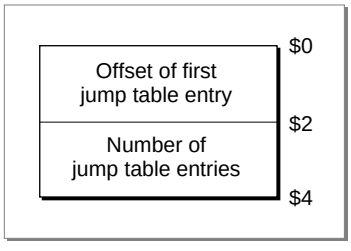
After transforming the jump table entries, the Segment Manager then calls the actual routine by executing the instruction in the last 6 bytes of the (now loaded) jump table entry. Any subsequent calls to the routine also execute this instruction.

Note that the last 6 bytes of the jump table entry are executed whether the segment is loaded or not. The effect of the instruction depends on the state of the entry at the time.

The jump table entries remain in their loaded state unless you call the `_UnloadSeg` routine, which restores them to their unloaded state.

Note that to set all the jump table entries for a segment to their loaded or unloaded state, the Segment Manager needs to know where in the jump table all the entries are located. It gets this information from the **segment header.** The segment header, which is 4 bytes long for the near model environment, contains the offset of the first routine's entry from the start of the jump table (2 bytes) and the number of entries for the segment (2 bytes). Figure 10-6 shows the segment header.

Figure 10-6 Near model segment header



Note
The segment header is different for the far model environment. See "The Far Model Segment Header Structure" (page 10-23), for more information. ◆

Bypassing MC68000 Addressing Limitations

68K compilers typically generate PC-relative instructions for intrasegment references. This restricts the size of segments to 32 KB because the PC-relative instructions on the MC68000 processor use a 16-bit offset. Similarly, references to addresses expressed as offsets from the address stored in A5 are also limited to 16-bit offsets on the MC68000 processor.

Since references to the jump table are expressed as positive offsets from A5, this effectively limits the size of the jump table to 32 KB. References to global variables are expressed as negative offsets from A5, so the size of the global data area is limited to 32 KB as well.

In the past, the Resource Manager used to limit resources to 32 KB, so 16-bit offsets were guaranteed to be sufficient.

Table 10-1 summarizes existing MPW solutions to these limitations. The sections that follow provide detail on how to implement these solutions. The section "32-Bit Everything" (page 10-17) describes a mechanism that allows you to remove all three limits. Which solution you choose depends on the specific needs of your program.

Note
Other development environments may use different methods to work around the 16-bit addressing limitations. ◆

In general, it is recommended that if you need to remove only one of the limits, you use the solution given for that limit. If you need to remove two or more limits, the 32-bit everything solution is probably your best choice.

Table 10-1 Classic 68K runtime architecture limits and solutions

Problem	Solution	Restrictions and effect on performance
Globals > 32 KB	SC/SCpp `-model farData` option	No restrictions. Code is bigger. Must link with `-model far` option.
	Use assembly language for 32-bit references	No restrictions.
Segment > 32 KB	SC/SCpp `-bigseg` option	Restricted to single-segment C programs running on 68020 and or higher CPU.
	ILink `-br 68k` option	No restrictions.
	ILink `-br 020` option	Program must run on a 68020 or higher.

continued

Table 10-1 Classic 68K runtime architecture limits and solutions (continued)

Problem	Solution	Restrictions and effect on performance
Jump table > 32 KB	ILink `-wrap` option	No restrictions. It decreases memory available for global data.
Everything > 32 KB	See the section "32-Bit Everything" (page 10-17).	No restrictions. Code is up to 30% bigger.

Increasing Global Data Size

To permit your application to use more than 32 KB of global data, you have the following options:

- Use the `-model farData` option when you compile C files that reference far data. See "Expanding Global Data and the Jump Table" (page 10-19) for additional information.
- Implement 32-bit references in assembly language when necessary.

When linking files compiled with the `-model farData` option, ILink sorts data modules into near and far groups by default, placing all 16-bit referenced global data as close to A5 as possible and all 32-bit referenced data farther away. Thus, any data with a 16-bit reference is forced to within 32 KB of A5 if possible.

If you are using assembly language, you must explicitly code 32-bit references when you want to avoid fixing a data module to within 32 KB of A5. For the MC68000, you could write something like this:

```
            IMPORT      LONGDATA:DATA
            MOVE.L      indirect(PC),D0         ; [4/7/9 clocks]offset -> scratch
                                                ; register
            MOVE.x      (A5,D0.L),dest          ; [ea: 3/6/7 clocks]access var
                                                ; (PEA,etc.)
            …

indirect:   DC.L        LONGDATA;               ;   32-bit offset of data
```

In code that is intended to run only on a 68020 microprocessor, you can do this:

```
MACHINE         MC68020
IMPORT          LONGDATA:DATA
MOVE.x          ((LONGDATA).L,A5),dest      ; move to destination
                                            ; (or PEA)
                                            ; [ea: 11/15/25 clocks]
```

The 68020 code, while smaller, runs more slowly than the 68000 code shown above if you ignore the possible impact of the temporary register required (11 versus 7 clock cycles best case, 15 versus 13 clocks cache case, and 25 versus 16 clocks worst case). Also note that the operand addressing mode shown in the last instruction uses normal 68000 syntax; it does not, in this instance, represent far model syntax.

Increasing Segment Size

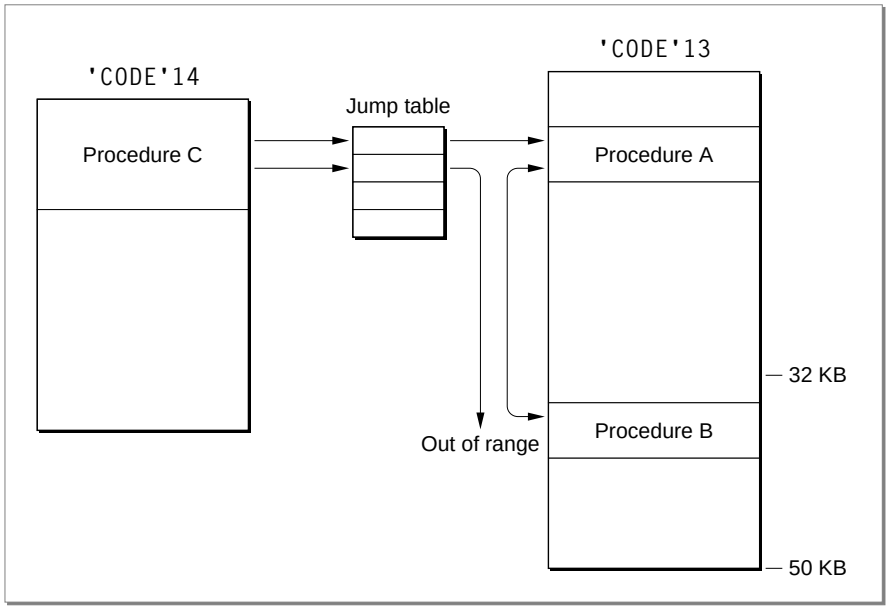
There are two methods for increasing segment size:

- You can use the `-bigseg` compiler option. This causes function calls within the same segment to be encoded with the `BSR.L` instruction (available on 68020 or higher CPUs), which is a PC-relative instruction with a 32-bit offset. This solution is right for single-code segments like command extensions (type `'XCMD'`) written in C. It does not work on 68000 machines.
- You can use the `-br 68k` or `-br 020` option of the ILink command. ILink then inserts small assembly-language modules called **branch islands** that transmit calls between two distant modules. The original call is modified to be a `JSR` instruction to the branch island, and the latter contains instructions to branch to the desired target.

Note
If the program you are writing is intended to run on a 68020 or higher CPU, you can use the `-br 020` option. This reduces code size and improves execution speed. ◆

Creating branch islands solves intrasegment reference problems, but is not a complete solution in the case where a routine located beyond the 32 KB limit is externally referenced. Figure 10-7 shows two segments, one of which is larger than 32 KB.

Figure 10-7 Branch islands and intersegment references



As you can see, the only reference that cannot be resolved is that to procedure B if it is made through the jump table. The ILink tool automatically tries to place externally referenced routines in the first 32 KB of a segment, but if this is not possible, it generates a linker error. In such cases, you should resegment your code or build your program with the `-model far` (32-bit everything) option.

Increasing the Size of the Jump Table

To increase the size of the jump table, use the `-wrap` option of the ILink command. This increases the memory allocated for the jump table at the expense of memory reserved for global data. In effect, this puts some of the jump table at negative offsets from A5.

This method is particularly useful for MacApp programs because they make little demand on global data space. However, at best, this method can only double the jump table size.

If you choose this option, intersegment calls, which are always routed through the jump table, might look like global references, as in this example:

```
JSR -48(A5)
```

The instruction used, `JSR` or `BSR`, makes it plain that it is not a global variable that is being referenced.

32-Bit Everything

The 32-bit everything method allows you to remove limitations on segment size, global data size, and jump-table size by using compiler and linker `-model far` options instead of the default value, which is `-model near`. For each compilation unit, the compiler allows you to choose

- full 32-bit offsets for global data by specifying the `-model farData` option
- full 32-bit offsets for code references by specifying the `-model farCode` option
- full 32-bit offsets for data and code by specifying the `-model far` option

You can link any combination of near and far model compiled modules, but if any of the modules are compiled with the `-model far`, `-model farData`, or `-model farCode` options, you must specify the `-model far` linker option.

▲ **WARNING**
Because the 32-bit everything solution is implemented by modifications to the `LoadSeg`, `UnloadSeg`, `Launch`, `Chain`, and `ExitToShell` traps, it will not work if your application patches these traps without calling the original traps when your patch completes. If you need to use `_LoadSeg` or `_UnloadSeg` in the 32-bit everything environment, you must use the routines in the `RTLib.o` library. For details, see Appendix B. ▲

In assembly language, the use of a 32-bit reference for the target address of an instruction must be explicitly demanded by use of the absolute long address syntax (*expr*).`L`, where *expr* is a relocatable expression. Two other requirements must be met:

- The relevant operand symbol must be imported. This means that the defining occurrence of the symbol must be in a different module than the module or modules containing its use as a 32-bit reference.

- The option `-model far` must be used for the assembly. Since the absolute long address syntax specifies absolute operands by definition, the use of this form with a relocatable symbol is an error unless you specify the `-model far` option.

Global data references, references to code in the same segment, and references to code in a different segment all cause the assembler to produce similar records that tell the linker that a 32-bit patch is needed. The linker determines whether the references are to code or data. If the reference is to code, the linker can also determine whether the reference is internal or external.

The example shown in Listing 10-1 illustrates using 32-bit references for the target address of an instruction.

Listing 10-1 Using 32-bit references for the target address of an instruction

```
            MAIN
            IMPORT STUFF                    ; Symbols from other
            IMPORT THERE                    ; modules must be
            IMPORT ELSEWHERE                ; imported.
            JSR (THERE).L                   ; Symbols are written using
            JSR (ELSEWHERE).L               ; (xxx).L syntax.
            ADD.W (STUFF).L,DO
            ENDMAIN

            PROC                            ; Note that THERE is in the MAIN
            EXPORT              THERE       ; segment
THERE       NOP
            ENDPROC
            SEG                 'SG1 '      ; Note that ELSEWHERE
            PROC                            ; is in a different segment
            EXPORT              ELSEWHERE
ELSEWHERE   NOP
            ENDPROC
            PROC
            DATA
            EXPORT              STUFF
STUFF       DS                  1
            ENDPROC
            END
```

How 32-Bit Everything Is Implemented

The implementation of the 32-bit everything solution affects the way global data, intrasegment references, and intersegment references are generated by the compiler and relocated by the linker. This section describes the changes that result from using this option. If you are using a low-level debugger or if your application depends on walking through jump-table entries, you need to be familiar with the details of this implementation.

Expanding Global Data and the Jump Table

Because jump-table entries and global data are both referenced relative to A5, far references to global data and to jump table entries are handled in a similar way.

If you compile and link units with any option that specifies the far model for data, any instruction that references global data is generated with a 32-bit absolute address. This address is the byte offset of the data item relative to the address stored in A5. The address of any instruction that references global data is stored in compressed form in an area called *A5 relocation information.* The modified `_LoadSeg` trap, using this information and the address stored in A5 at load time, relocates each instruction during loading by subtracting the 32-bit address field of the instruction from the value of A5.

If you compile and link units with any option that specifies the far model for code, any `JSR` instruction that references a jump-table entry is generated with a 32-bit absolute address. The address of any instruction that makes such a reference is recorded in compressed form in the A5 relocation information area. The modified `_LoadSeg` trap adds the value of A5 to the address fields of the `JSR` instruction at load time.

Note that because intersegment references linked under this model appear in the disassembled code as `JSR` instructions to an absolute address, it is no longer obvious that you are going through the jump table. If the value of the address is greater than that in A5, it is possible you are going through the jump table.

For additional information about A5 relocation information, see the section "The Far Model Segment Header Structure" (page 10-23).

Intrasegment References

If you compile and link units with any option that specifies the far model for code, any instruction that makes an intrasegment reference is generated with a 32-bit absolute address. This address is the byte offset from the beginning of the segment to the referenced entry point. The address of any instruction making such a reference is stored in compressed form in an area called **segment relocation information.** The modified `_LoadSeg` trap, using this information and the load address of the segment, relocates each such instruction by adding the load address of the segment to the instruction's 32-bit address field.

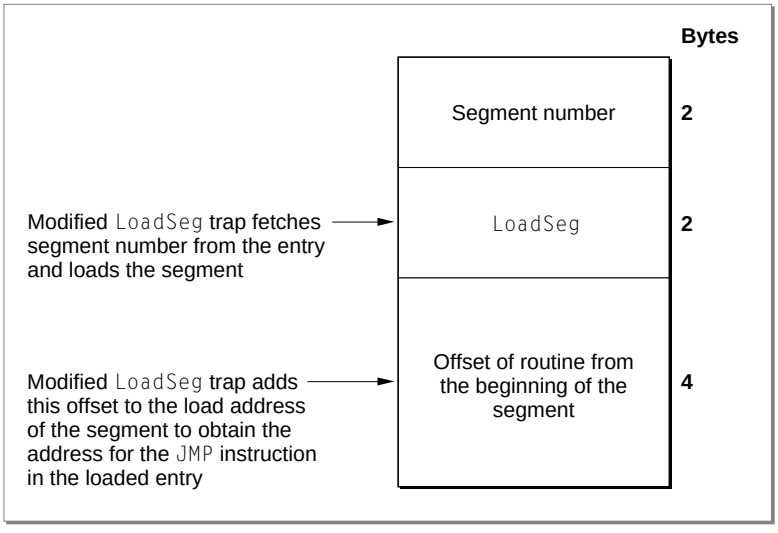
The Far Model Jump Table

Compilation and linking with the option `-model far` results in a change in the format of the jump table and a change in the format of unloaded entries.

- Because a segment can be larger than 32 KB, 4 bytes are required to describe the offset of a routine from the beginning of its segment. As the model near unloaded entry for a routine allowed only 2 bytes to specify the offset of the routine, this requires a change in the unloaded format of jump table entries.
- Because segments compiled with the option `-model near` can be linked with segments compiled with the option `-model far`, jump-table entries for the same segment are not necessarily contiguous.

Figure 10-8 shows the unloaded jump table entry for a routine in a segment that is linked using the far model. As you can see, the modified entry omits the instruction that puts the segment number on the stack. The 2 bytes saved are then used to store the larger 4-byte offset that locates the routine within the segment. The Segment Manager gets the segment number from the entry itself and loads that segment.

Figure 10-8 Far model unloaded jump table entry

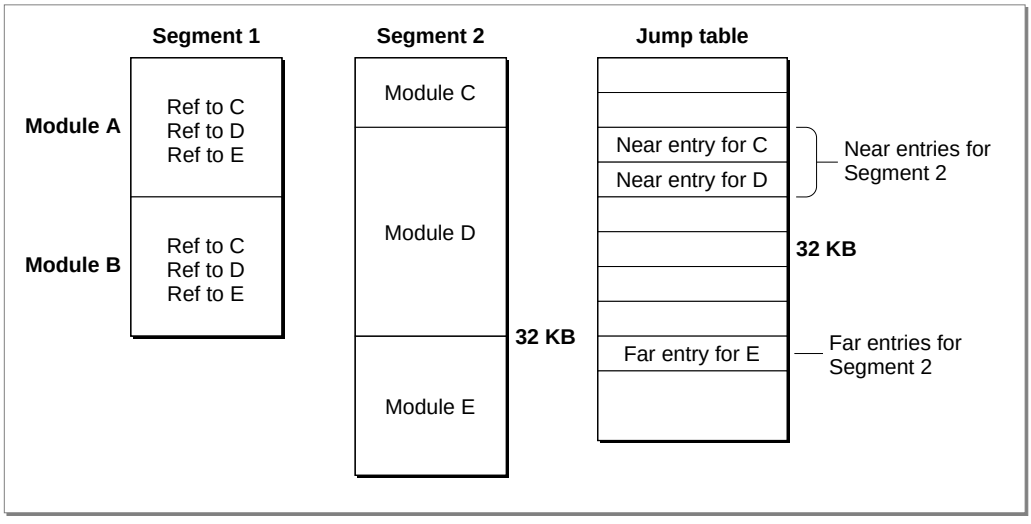


In the standard near model jump table, entries for routines in the same segment are stored contiguously. In a jump table created for a program linked under the far model, entries for routines in the same segment might not be contiguous. Consider the case shown in Figure 10-9. When the linker builds segment 1 and segment 2, it places code compiled with `-model near` within the first 32 KB and code with `-model far` beyond the 32 KB limit.

When the jump table is built, the linker places near-referenced entries within the first 32 KB; far-referenced entries are placed after all near references. Thus near and far references for the same segment can be stored in different areas of

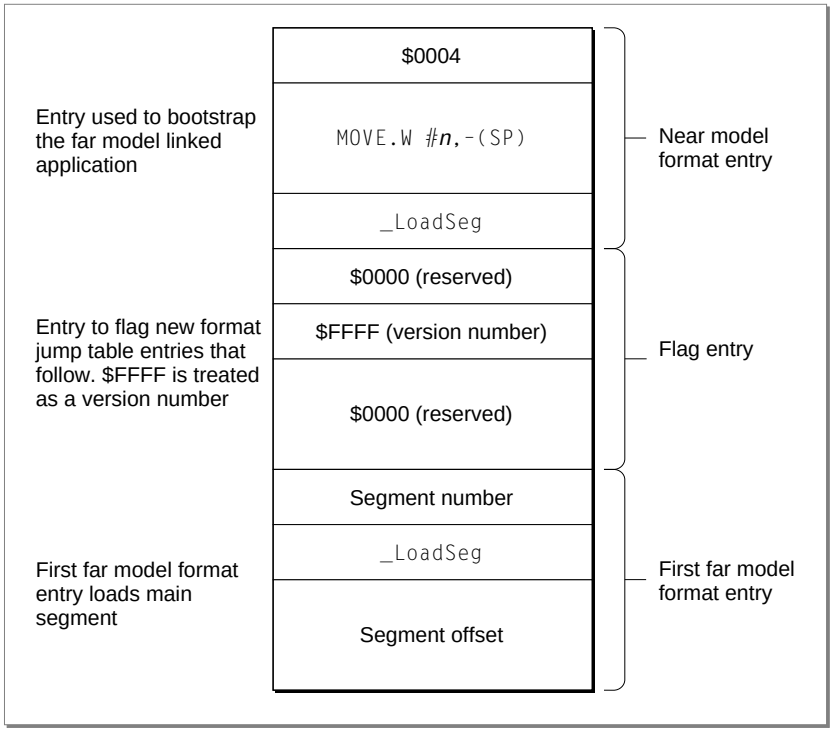
the jump table. In Figure 10-9, the entry for `Module E` is not contiguous with near entries for the other modules contained in segment 2.

Figure 10-9 Separation of near and far references in the far model jump table



The format of the jump table built for programs linked under the far model is different from that for programs built under the near model. Figure 10-10 shows the format of the far model jump table.

Figure 10-10 The far model jump table structure



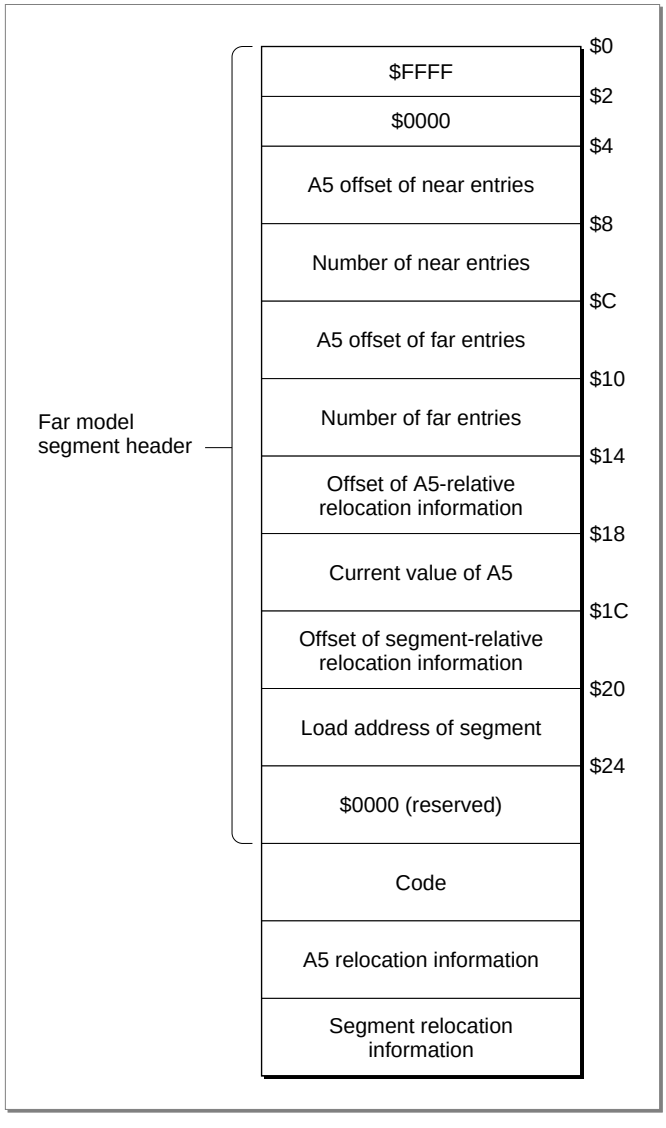
The first entry in the jump table is a near model format entry used to load a segment that patches the `_LoadSeg` trap, segment *n*. The next entry is an entry used to flag the far model format jump table. The third entry is a far model format entry. Remember that what's different about it affects only the information stored for its unloaded state. This third entry is used to load segment 1, which is the segment containing the program's main entry point.

The Far Model Segment Header Structure

Near model segments have a 4-byte header that provides the information required by the Segment Manager to transform jump table entries from their unloaded state to their loaded state. Segments linked with the `-model far` option have a larger header and contain relocation information. The format of

the far model segment header is shown in Figure 10-11. Following the header are the code, the A5 relocation information, and the segment relocation information.

Figure 10-11 The far model segment header



The meaning of each field in Figure 10-11 is as follows.

Address	Entry
$0	This field determines whether the segment has been built according to the far model option. Namely, the first word of the segment header must match the version field in the jump-table flag entry, $FFFF.
$2	Reserved.
$4	The byte offset from A5 of the first near model jump table entry.
$8	The number of near entries.
$C	The byte offset from A5 of the first far model jump table entry.
$10	The number of far model entries.
$14	The offset (from the beginning of the segment) of the relocation information for A5-relative references. A5-relocation information contains the addresses of all instructions in a segment that references far model global data or far model jump table entries.
$18	The current A5 value, which is added to the offset specified in the A5-relative address field of the instruction to calculate the actual address.
$1C	The offset, from the beginning of the segment, of the relocation information for PC-relative references.
$20	The segment load address, which is added to the offset specified in the A5-relative address field of the instruction to calculate the actual address of the entry point.
$24	Reserved.

Relocation Information Format

Relocation information consists of a consecutive list of offsets between longwords that need to be relocated at load time, beginning with the offset of the first such longword from the start of the segment.

Some data compression is used in recording this information. Because instructions start at even addresses, it suffices to record the offset values

divided by two. In Table 10-2, the various encodings are shown as bit strings. The part of the value represented by "bbb..." gives, when doubled, the desired offset value.

Table 10-2 Relocation information

Relocation item	Interpretation
00000000 00000000	End of relocation information
0bbbbbbb	Offsets between $02 and $FE
1bbbbbbb bbbbbbbb	Offsets between $0100 and $FFFE
00000000 1bbbbbbb bbbbbbbb bbbbbbbb bbbbbbbb	Offsets between $00010000 and $FFFFFFFE

CHAPTER 11

Classic 68K Runtime Conventions

Contents

This chapter covers data storage and parameter-passing conventions for the classic 68K runtime environment. Classic 68K conventions can vary depending on the programming language and the compiler you use; this chapter assumes you are using the SC/SCpp compiler and C or Pascal calling conventions.

Data Types

Table 11-1 lists the various binary data types and their sizes in the classic 68K runtime environment.

Table 11-1 Data types in the classic 68K runtime environment

Type	Size (bytes)	Alignment (bytes)	Range	Notes
`UInt8`	1	1	0 to 255	
`SInt8`	1	1	–128 to 127	
`SInt16`	2	2	–32,768 to 32,767	
`UInt16`	2	2	0 to 65,535	
`SInt32`	4	2	-2^{-31} to $2^{31}-1$	
`UInt32`	4	2	0 to $2^{32}-1$	
Boolean	1	1		0 = false, nonzero = true
`float`	4	2	$\pm(2^{-149}$ to $2^{127})$	IEEE 754 standard
`double`	8	2	$\pm(2^{-1074}$ to $2^{1023})$	IEEE 754 standard
Pointer	4	2	0 to FFFFFFFF	
`extended`	10 or 12	2		SANE or MC68881 data type

All numeric and pointer data types are stored in big-endian format (that is, high bytes first, then low bytes). Signed integers use two's-complement representation.

IMPORTANT

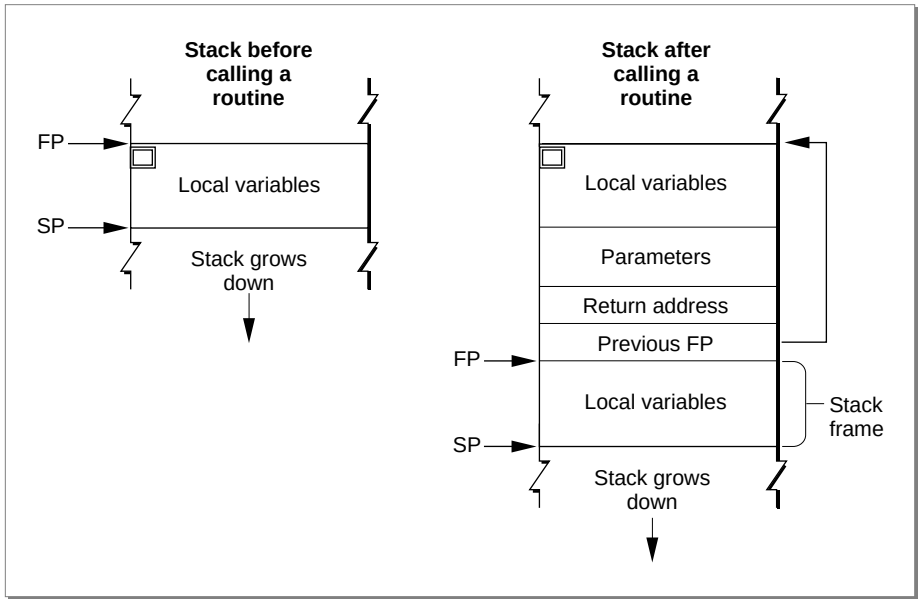
The layout of the `extended` data type is either that of the SANE 80-bit data type or that of the 96-bit MC68881 data type, depending on the software development environment used. ▲

The size of data structures and unions must be a multiple of two (2-byte alignment), and an extra byte may be added at the end to meet this requirement. Items inside a data structure (except for types `UInt8` and `SInt8`) are placed on a 2-byte boundary with an extra padding byte inserted if necessary. Type `UInt8` and type `SInt8` items (single variables or arrays) are merely placed in the next available byte.

Classic 68K Stack Structure and Calling Conventions

The classic 68K runtime architecture uses a stack-based parameter-passing system, as shown in Figure 11-1.

Figure 11-1 A 68K stack frame before and after calling a routine



The stack grows from high-memory addresses towards low-memory addresses. The end that grows or shrinks is usually referred to the "top" of the stack, despite the fact that it is at the lower end of memory occupied by the stack.

The boundaries of the stack are defined by two pointers:

- The **stack pointer (SP),** which points to the top of the stack and defines its current downward limit. Operations that push data onto the stack or pop data off of it do so by adjusting the value of the stack pointer. The classic 68K runtime architecture uses the A7 register as the stack pointer.
- The **frame pointer (FP),** which points to the base in memory of the current **stack frame,** the area of the stack used by a routine for its parameters, return address, local variables, and temporary storage. By keeping track of the frame pointer value, the operating system can find the beginning of the stack frame when it has to pop data off the stack. The classic 68K runtime architecture uses the A6 register as the frame pointer.

Parameters passed by a routine are always placed on the stack above the frame pointer, while local variables are always placed below the frame pointer. Data passed onto the stack is always aligned to 2 bytes. If you pass a single-byte parameter (such as a single character), a padding byte is added by decrementing the stack pointer by 2 bytes instead of 1 (the padding byte is the most significant byte).

The classic 68K runtime environment supports many non-standard calling conventions. For example, C calling conventions can vary depending on the type of call (some system calls have their own conventions) and the development environment. However, in C you can specify Pascal conventions for a routine by using the `pascal` keyword. Pascal calling conventions are standardized and supported in all 68K development environments.

For example, the routine declared in C as

```
int mooFunc(UInt8, double);
```

uses C calling conventions, while

```
pascal int mooFunc(UInt8, double);
```

uses Pascal calling conventions.

Note
These parameter passing conventions are part of Apple's standard for procedural interfaces. Object-oriented languages may use different rules for their own method calls. For example, the conventions for C++ virtual function calls may be different from those for C functions. ◆

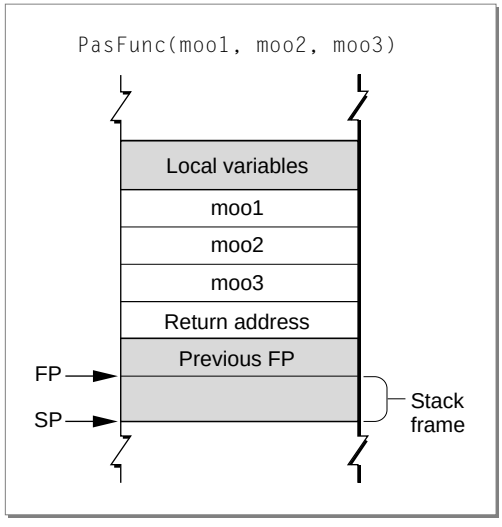
Pascal Calling Conventions

When following Pascal calling conventions, the caller passes space for the return value before pushing any parameters. The caller then passes parameters from left to right. For example, given the code

```
cow = PasFunc(moo1, moo2, moo3);
```

the calling routine first pushes the value of `moo1` onto the stack, followed by `moo2` and then `moo3` as shown in Figure 11-2.

Figure 11-2 Passing parameters onto the stack in Pascal



Pascal allows only a fixed number of parameters to be passed to the called routine. However, this means the size of the stack frame can be determined at compile time, so the called routine assumes responsibility for deallocating (popping) parameters before returning.

Function values are returned on the stack, as follows:

- If the value is 4 bytes or smaller in size, the item on the stack is the return value.
- If the return value is larger than 4 bytes, the item on the stack is a pointer to the return value.

The calling routine must allocate space on the stack for the return value before pushing any parameters, and the same routine is responsible for popping the result after the call.

SC Compiler C Calling Conventions

As mentioned earlier, the classic 68K runtime environment supports several different C calling conventions. This section describes the C calling conventions used by the SC compiler in the MPW development environment.

C allows either a fixed or variable number of parameters to be passed to the called routine. In an ANSI-style C syntax definition, a routine with a variable number of arguments typically appears with ellipsis points (…) at the end of its input parameter list.

A variable-argument function may have several required (that is, fixed) parameters preceding the variable parameter portion. For example, the function definition

```
mooColor(number, [color1...])
```

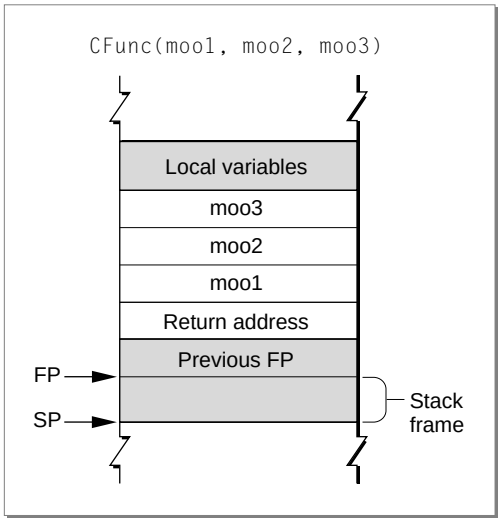
gives no restriction on the number of *color* arguments, but you must always precede them with a *number* argument. Therefore, *number* is a fixed parameter.

Parameters passed by routines are pushed onto the stack from right to left. For example, given the code

```
cow = CFunc(moo1, moo2, moo3);
```

the calling routine first pushes the value of `moo3` onto the stack, followed by `moo2` and then `moo1`, as shown in Figure 11-3.

Figure 11-3 Passing parameters onto the stack in C



The return address of the routine is the last item pushed onto the stack.

The calling function is responsible for parameter deallocation (that is, popping parameters off the stack) after the called routine has returned. If the called routine is a function, the function value is normally returned in register D0 (or, for floating-point values, in register F0). In the case of data structures or values larger than 4 bytes, however, the caller must allocate space for the return value and pass a pointer to that storage space as the first (that is, the leftmost) parameter.

Register Preservation

Table 11-2 lists registers used in the classic 68K runtime environment and their volatility in routine calls. Registers that retain their value after a routine call are called *nonvolatile*. Note that these register conventions are for C and Pascal-style calls. Certain system calls may use different conventions, so you should check their definitions in the appropriate *Inside Macintosh* book before using them. All registers are 4 bytes long.

Table 11-2 Volatile and nonvolatile registers

Type	Register	Preserved by a function call?	Notes
Data register	D0 through D2	No	
	D3 through D7	Yes	
Address register	A0	No	
	A1	No	
	A2 through A4	Yes	
	A5	See note	Used to access global data objects and the jump table.
	A6	Yes	Used as the frame pointer, which points to the base of the stack frame.

continued

Table 11-2 Volatile and nonvolatile registers (continued)

Type	Register	Preserved by a function call?	Notes
	A7	See note	A7 is the stack pointer used to push and pop parameters and other temporary data items.
Floating-point register	F0 through F3	No	When present.
	F4 through F7	Yes	When present.
Condition Register	CR	No	Bits are set by compare instructions and used for conditional branching.

Appendixes

Terminology Changes

Some of the terminology, constant names, and data type names in this book have been changed from previous documentation. The lists here describe the terminology used as of the E.T.O. 21 software development release. In some cases, the terminology has changed due to conceptual shifts, with the result that the old term and the new term are not directly interchangeable. Such changes are explained in the notes.

Table A-1 lists old terms used in previous documentation and the current terms used in this book.

Table A-1 Changes to terminology

Old term	New term	Notes
A5 world	Direct data area	This change applies only to the CFM-68K runtime environment.
Global data world	Direct data area	Direct data area entries can contain either the data itself or a pointer to indirect data.
Table of Contents	Direct data area	The old Table of Contents is the set of pointers in the direct data area that point to indirect data.
Table of Contents Register (RTOC)	Base register	Also referred to as simply GPR2.
XDataPointer	See next column	XDataPointers are now simply referred to as pointers to indirect data.
XPointer	See next column	XPointers are now simply referred to as the pointer to the transition vector.
XVector	Transition vector	Changed to emphasize the commonality between the PowerPC and CFM-68K implementations.

Table A-2 lists names used in previous versions of the `codeFragments.h` header file and the new names.

Table A-2 Changes to names in the `CodeFragments.h` header file

Old name	New name	Notes
`LoadFlags`	`kCFragLoadOptions`	Possible values for this flag are `kReferenceCFrag`, `kFindCFrag`, and `kPrivateCFragCopy`.
`kFindLib`	`kFindCFrag`	
`kFullLib`	`kIsCompleteCFrag`	
`kInMem`	`kMemoryCFragLocator`	
`kIsApp`	`kApplicationCFrag`	
`kIsDropIn`	`kDropInAdditionCFrag`	Drop-in additions are now called *plug-ins*.
`kIsLib`	`kImportLibraryCFrag`	
`kLoadCFrag`	`kReferenceCFrag`	
`kLoadLib`	`kReferenceCFrag`	
`kLoadNewCopy`	`kPrivateCFragCopy`	
`kMotorola68K`	`kMotorola68KCFragArch`	
`kMotorola68KArch`	`kMotorola68KCFragArch`	
`kNewCFragCopy`	`kPrivateCFragCopy`	
`kOnDiskFlat`	`kDataForkCFragLocator`	
`kOnDiskSegmented`	`kResourceCFragLocator`	
`kPowerPC`	`kPowerPCCFragArch`	
`kPowerPCArch`	`kPowerPCCFragArch`	
`kPrivateConnection`	`kPrivateCFragCopy`	
`kWholeFork`	`kCFragGoesToEOF`	

Table A-3 lists older "source code" data types and the binary counterparts used in this book. Note that these mappings assume you are using MPW compilers. Other development environments may assume different sizes (2 bytes for type `int` rather than 4, for example).

Table A-3 Changes to names of data types

Old type	New type
`char`	`UInt8`
`signed char`	`SInt8`
`short`	`SInt16`
`unsigned short`	`UInt16`
`int`	`SInt32`
`unsigned int`	`UInt32`
`long`	`SInt32`
`unsigned long`	`UInt32`

The RTLib.o and NuRTLib.o Libraries

This appendix describes the interface to the MPW RTLib libraries `RTLib.o` and `NuRTLib.o`, which allow access to the Segment Manager routines in the classic 68K far model (32-bit everything) and CFM-68K runtime environments respectively. These routines are useful if an application needs to have knowledge of its execution environment or if it needs to have knowledge of *another* application's environment. Specifically, if your application needs to patch the `_LoadSeg` trap or the `_UnloadSeg` trap in the far model or CFM-68K runtime environments, you need to use the RTLib libraries.

The interface for the RTLib routines is identical for both far model and CFM-68K runtime environments. The only difference is that you must link to different libraries when you build your application.

Note
Traditional classic 68K near model applications can patch the `_LoadSeg` and `_UnloadSeg` traps normally without having to go through the RTLib routines. ◆

IMPORTANT
The RTLib libraries can be used only for calls from a segmented application. CFM-68K runtime shared libraries cannot use the `NuRTLib.o` library, and any A5 or fA5 referenced calls made to a nonsegmented A5 world returns an error. ▲

Runtime Interface

The RTLib runtime interface consists of a single procedure call (as defined in `RTLib.h`):

```
pascal OSErr Runtime (RTPB* runtimeParams);
```

The RTPB data type is a structure in which you specify one of four posssible parameter blocks:

```
struct RTPB {
    short fOperation;
    void* fA5;
    union {
        RTGetVersionParam fVersionParam;
        RTGetJTAddrParam fJTAddrParam;
        RTSetSegLoadParam fSegLoadParam;
        RTLoadSegbyNumParam fLoadbyNumParam;
        } fRTParams;
    };
typedef struct RTPB RTPB;
```

The fields of the RTPB structure are as follows:

- The fOperation field indicates the type of operation to be performed, and it can be set to any value shown in Table B-1 (page B-3). See "Runtime Operations" (page B-4) for more detailed information.
- Any operation whose name ends in A5 requires a value for the fA5 field, which indicates the address of an A5 world. The similarly named operation without the A5 suffix uses the current value of A5 for this parameter.
- The fRTParams field is a parameter block consisting of one of four structures that hold the parameters for the appropriate operation. See the descriptions for each operation in "Runtime Operations" (page B-4) for more details about these structures.

Table B-1 `Runtime` routine operation values

Value		Description
`kRTSetPreLoad`	`kRTSetPreLoadA5`	Arranges for a user handler to be called by the Segment Manager before loading a segment.
`kRTSetSegLoadErr`	`kRTSetSegLoadErrA5`	Arranges for a user handler to be called if a segment load fails.
`kRTSetPostLoad`	`kRTSetPostLoadA5`	Arranges for a user handler to be called by the Segment Manager immediately after loading a segment.
`kRTSetPreUnload`	`kRTSetPreUnloadA5`	Arranges for a user handler to be called by the Segment Manager before unloading a segment.
`kRTGetVersion`	`kRTGetVersionA5`	Returns version number of A5 world.
`kRTGetJTAddress`	`kRTGetJTAddressA5`	Returns address of the code pointed to by the specified function address.
`kRTPreLaunch`		Required if you need to call the `_Launch` or `_Chain` routine.
`KRTPostLaunch`		
`KRTLoadSegbyNum`	`KRTLoadSegbyNumA5`	Loads a segment by segment number (CFM-68K only).

The `Runtime` routine can return an error value as shown in Table B-2.

Table B-2 `Runtime` routine error values

Error	Description
`eRTNoErr`	No error (success)
`eRTInvalidOP`	Invalid operation
`eRTBadVersion`	Invalid version
`eRTInvalidJTPtr`	Invalid jump table pointer
`eRT_not_segmented`	A5 world not segmented (for example, the A5 world of a CFM-68K shared library)

Runtime Operations

This section describes the operations in Table B-1 in more detail. Note that operations are grouped according to which structure they use in the `fRTParams` field of the `RTPB` structure.

Segment Manager Hooks

Several `Runtime` operations allow the application to take control and execute a user-defined handler routine during the segment loading or unloading process. These operations are

- `kRTSetPreLoad` and `kRTSetPreLoadA5`, which pass control to the handler before loading a segment
- `kRTSetSegLoadErr` and `kRTSetSegLoadErrA5`, which pass control to the handler if the segment load fails
- `kRTSetPostLoad` and `kRTSetPostLoadA5`, which pass control to the handler after loading a segment
- `kRTSetPreUnload` and `kRTSetPreUnloadA5`, which pass control to the handler before calling `_UnloadSeg`

In each of these cases, control is passed by replacing the null user vector (set up by the patched Segment Manager) with a user handler.

The `fRTParams` structure used with these operations is as follows:

```
struct RTSetSegLoadParam {
    SegLoadHdlrPtr fUserHdlr;
    SegLoadHdlrPtr fOldUserHdlr;
    };
typedef struct RTSetSegLoadParam RTSetSegLoadParam;
```

The pointer `fUserHdlr` points to the user handler to be called at the time indicated by the operation. A pointer to the original (bypassed) handler is returned in `fOldUserHdlr`.

User Handlers

A user handler is defined as follows:

```
typedef pascal short (*SegLoadHdlrPtr)(RTState* state)
```

The handler may return a result code of type `short`. This code is ignored by the Segment Manager except in the case of the error handler. See "Error Handling With kRTSetSegLoadErr" (page B-7) for more details.

▲ **WARNING**
User handlers must be defined within the segment to be loaded into memory when the handler is invoked (usually the main segment). Also, the user handler must not call any routines in unloaded segments as this may result in a system crash. ▲

Information about the Segment Manager operation is passed to the user handler through the `RTState` structure. This structure has the following form:

```
struct RTState {
    unsigned short   fVersion;  /* runtime version */
    void*     fSP;              /* SP: address of user return address */
    void*     fJTAddr;          /* PC: address of jump table entry */
                                /*      or  (see fCallType) */
                                /*      address of a transition vector*/
    long      fRegisters[15];   /* registers D0-D7 and A0-A6 */
```

```
    short     fSegNo;           /* segment number */
    ResType   fSegType;         /* segment type (normally 'CODE') */
    long      fSegSize;         /* segment size */
    Boolean   fSegInCore;       /* true if segment is in memory */
    Boolean   fCallType;        /* 0 = _LoadSeg, */
                                /* 1 = fJTAddr, address of TVector */
    OSErr     fOSErr;           /* error number */
    long      fReserved2;       /* (reserved for future use) */
    };
typedef struct RTState RTState;
```

The fields in the structure are as follows:

- `fVersion` is the version number from the A5 or fA5 runtime world.
- `fSP` has the value of the stack pointer when either `_LoadSeg` or `_UnloadSeg` was executed.
  - In the case of `_LoadSeg`, if the jump table entry was reached using a `JSR` instruction, `fSP` is a pointer to the user return address. You can modify the stack pointer value within an error handler to change the return address if you want to continue execution after trapping an error. See "Error Handling With kRTSetSegLoadErr" (page B-7) for more information. However, this is not recommended since there may not be a user return address on the stack.
  - In the case of `_UnloadSeg`, `fSP` points to the return adress from the `_UnloadSeg` call.
- `fJTAddr` points to either a jump table entry or a transition vector depending on the runtime environment and the value of `fCallType`:
  - In a `_LoadSeg` call (`fCallType` is `0`), `fJTAddr` points to the jump table entry called by the user code prior to the `_LoadSeg` call. You can modify the value of `fJTAddr` within an error handler if you want to retry the segment load procedure.
  - In an `_UnloadSeg` call, `fJTAddr` points to the function address passed to `_UnloadSeg`.
  - In the CFM-68K runtime environment, the `fJTAddr` field is always the address of a transition vector. You cannot modify this field when `fCallType` is `1`.

  Note that you should not make any assumptions about the layout of the jump table entry since it varies between the far model and CFM-68K runtme environments and may change in the future.

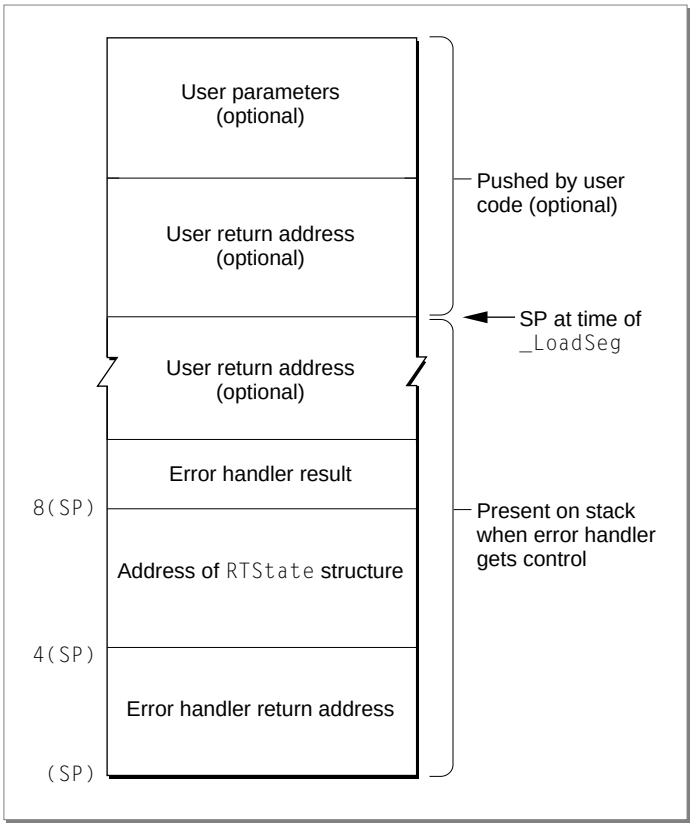
- `fRegisters` is an array of long integers that contains the register values at the time `_LoadSeg` was called. The registers are saved in the order D0 through D7, then A0 through A6.
- `fSegType` and `fSegNo` contain the segment's resource type and ID. `fSegType` is usually `'CODE'` but this may change in the future.
- `fSegSize` contains the size of the segment, in bytes.
- `fSegInCore` indicates whether the segment is in memory. If `fSegInCore` is `true`, the segment is already in the heap but has not been locked. (If the segment is resident, no memory needs to be allocated for it.)
- The `fCallType` field is used by other fields whose meanings are dependent on how the segment load was invoked. If `fCallType` is `1`, the segment load was invoked through a function call by a pointer (or by a virtual method dispatch in C++).
- `fOSErr` contains an error number. This field is valid only when this structure is passed to an error handler.

All attempts to modify the `RTState` structure are ignored except for alterations of `fJTAddr` by the user error handler.

Error Handling With kRTSetSegLoadErr

When `kRTSetSegLoadErr` invokes the user error handler (that is, when a segment loading error occurs), the stack has the form shown in Figure B-1.

Figure B-1 The stack when a user error handler is called



The error handler should use the information at the following locations:

- The word at `8(SP)` is reserved for the error handler's action code (as described later in this section).
- The value at `4(SP)` points to the `RTState` structure, which contains information about the error.
- The value at `(SP)` is the return address from the error handler. This value may or may not be used depending on how the routine handles the error.

Items on the stack labeled as optional may not actually appear. For example, a simple `JMP` instruction would not push user parameters or a return address onto the stack.

The error handler should examine the `RTState` structure and then take appropriate action (for example, release some memory). After doing so, the handler can do one of the following:

- Return an action code on the stack for the Segment Manager and then return. Current action codes are shown in Table B-3. Attempts to pass any other value to the Segment Manager results in the system error `daLoadErr`.
- use a `LONGJMP` (or the equivalent) instruction to pass control to another error handler set up in a parent stack frame. This second handler could save the current document, alert the end user, and quit the application.

Table B-3 Error handler action codes

Value	Action
`kRTRetry`	Retry. This action restores the stack to its state before the call to `_LoadSeg` and reexecutes the jump table entry. If no errors occur this second time, execution continues normally. If the handler modified `fJTAddr` in the `RTState` structure, execution resumes at the new address. Note that this technique can easily create an infinite loop if the segment loading attempt always fails. Your routine should include a retry counter to break out of the loop after a specified number of tries.
`kRTContinue`	Continue. This action restores the stack to its state before the `_LoadSeg` call and sets the program counter (PC) to the user return address in the stack. Note that this action is risky since a return address may not always be present on the stack.

kRTGetVersion and kRTGetVersionA5

The operation `kRTGetVersion` returns the value of the current A5 world and `kRTGetVersionA5` returns the value of the specified A5 world.

The `fRTParams` structure (page B-2) used with these operations is as follows:

```
struct RTGetVersionParam {
    unsigned short fVersion;
    };
typedef struct RTGetVersionParam RTGetVersionParam;
```

The `kRTGetVersion` operation assumes the current A5 world, while `RTGetVersionA5` lets you specify one in the `fA5` field of the `RTPB` structure (page B-2).

The `fVersion` field holds the returned version number as shown in Table B-4.

Table B-4 Current version numbers

Version number	Description
$0000	Classic 68K near model A5 world
$FFFD	CFM-68K runtime A5 world
$FFFF	Classic 68K far model (32-bit everything) A5 world

kRTGetJTAddress and kRTGetJTAddressA5

The operation `kRTGetJTAddress` returns the address of the code that the specified function address points to in the current A5, and `kRTGetJTAddressA5` does the same for a specified A5 world.

The fRTParams structure (page B-2) used with these operations is as follows:

```
struct RTGetJTAddrParam {
    void*fJTAddr;
    void*fCodeAddr;
    };
typedef struct RTGetJTAddrParam RTGetJTAddrParam;
```

- In the classic 68K runtime environment, fJTAddr is a function address. In the CFM-68K runtime environment, fJTAddr is the address of a transition vector.
- fCodeAddr contains the returned code address. If the segment is not loaded, fCodeAddr is set to 0.

The kRTGetJTAddress operation assumes the current A5 world, while RTGetJTAddressA5 lets you specify one in the fA5 field of the RTPB structure (page B-2).

kRTPreLaunch and kRTPostLaunch

In the classic 68K near model environment, you cannot call _Launch or _Chain directly, but must instead use the Process Manager call LaunchApplication (See *Inside Macintosh: Processes* for more details). If you need to call _Launch or _Chain under the far model environment, you must wrap a call to _Launch with calls to Runtime using the kPreLaunch and KPostLaunch operations as follows:

```
IMPORT(Runtime): CODE

MOVE.W#kRTPreLaunch,-(SP)   ; push fOperation
SUBQ.W#2,-(SP)              ; room for result
PEA 2(SP)                   ; push ptr to RTPB
JSR Runtime                 ; prepare for launch

_Launch                     ; attempt a launch

MOVE.W#kRTPostLaunch,-(SP)  ; push fOperation
SUBQ.W#2,-(SP)              ; room for result
PEA 2(SP)                   ; push ptr to RTPB
JSR Runtime                 ; post-launch housekeeping
```

The only field used in the RTPB structure (page B-2) is fOperation. Neither kPreLaunch or kPostLaunch require an fRTParams parameter block.

If you need to call _Chain, you must wrap the routine with kPreLaunch and kPostLaunch in the same manner as with the _Launch routine.

IMPORTANT

You must never call the _Chain trap since it is not implemented by the System 7 Process Manager. ▲

The CFM-68K runtime environment does not support calling _Launch or _Chain directly from a CFM-68K application. You can call kPreLaunch and kPostLaunch in the CFM-68K runtime environment, but the routines do nothing.

kRTLoadSegbyNum and kRTLoadSegbyNumA5

This operation (available only for CFM-68K) allows you to explicitly load a segment by segment number. kRTLoadSegbyNum assumes the current A5 world, while kRTLoadSegbyNumA5 lets you specify one in the fA5 field of the RTPB structure (page B-2).

No user vectors are called while attempting to load the segment. If for any reason the segment cannot be loaded, the operation returns OSErr.

The fRTParams structure (page B-2) used with these operations is as follows:

```
struct RTLoadSegbyNumParam{
    short fSegNumber;
    };
typedef struct RTLoadSegbyNumParam RTLoadSegbyNumParam;
```

The fSegNumber field holds the specified segment number. If there is insufficient memory to load the segment, the GetResource call returns a Memory Manager error. If fSegNumber is not a valid segment number, GetResource also returns an error.

A Preload Example

Listing B-1 shows a C program that installs a preload handler and uses it to print information about the segment.

To compile and link this example for the classic 68K far model environment, use the following MPW commands:

```
SC -model far example.c -o example.c.o -i {CIncludes}
ILink -model far -w -t MPST -c 'MPS ' -o example ∂
    example.c.o ∂
    {Libraries}RTLib.o ∂
    {Libraries}Interface.o ∂
    {Libraries}IntEnv.o ∂
    {Libraries}MacRuntime.o ∂
    {CLibraries}StdCLib.o ∂
```

For the CFM-68K runtime environment, use the following commands:

```
SC -model cfmseg example.c -o example.c.o -i {CIncludes}
Ilink -model cfmseg -xm e -w -t MPST -c 'MPS ' -o example ∂
    example.c.o ∂
    {CFM68KLibraries}NuRTLib.o ∂
    {CFM68KLibraries}NuMacRuntime.o ∂
    {SharedLibraries}InterfaceLib ∂
    {SharedLibraries}StdCLib
```

Listing B-1 A preload handler example

```
#include <stdio.h>
#include <types.h>
#include <RTLib.h>
#pragma segment One
one ()
{
    /*  do something */
}
```

```
#pragma segment Main
pascal short preload_handler(RTState* state)
{

    /*  print segment information */

    printf("segno= %d\n",state->fSegNo);
    printf("segtype= %.4s\n",&(state->fSegType));
    printf("segsize= %d\n",state->fSegSize);
    if (state->fSegInCore) printf("incore = yes\n");
    else printf("incore = no\n");
    return(0);
}
main ()
{
    RTPBparam_block, *p;
    OSErrerror;

    /*  load printf segment so that the preload handler does not */
    /*  invoke another call to _LoadSeg */

    printf("load printf segment\n");
    /*  load the handler */

    p = &param_block;
    p->fOperation = kRTSetPreLoad;
    p->fRTParam.fSegLoadParam.fUserHdlr = (void*)&preload_handler;
    error = Runtime(p);

    /*  load the segment */

    one();
}
```

Glossary

68K application An application that contains code only for a 68K microprocessor. Compare **fat application, PowerPC application.**

68K-based Macintosh computer Any computer containing a 680x0 central processing unit that runs the Mac OS. Compare **PowerPC-based Macintosh computer.**

68K microprocessor Any member of the Motorola 68000 family of microprocessors.

accelerated resource An executable resource consisting of a routine descriptor and PowerPC code that specifically models the behavior of a 68K stand-alone code resource.

A5 world In classic 68K and CFM-68K runtime programs, a memory partition that contains the QuickDraw global variables, the application global variables, the application parameters, and the jump table—all of which are accessed through the A5 register. Sometimes called the *global variable world.*

A-line instruction An instruction used to execute Toolbox and Operating System routines. The first word of an A-line instruction is binary 1010 (hexadecimal A). Also known informally as an *A-trap.*

application A program of type `'APPL'` that is launched from the Finder. Applications typically use event-driven programming and have a user interface. See also **program.**

application transition vector A 12-byte transition vector used in the CFM-68K runtime environment. The first two fields contain the address of a function and the value to be placed in A5 when the function executes. Because applications can be segmented, the third field contains information to locate the function within a particular segment. See also **shared library transition vector, transition vector.**

A-trap See **A-line instruction.**

base register The register that holds a reference address used to access a fragment's data area.

branch island A small assembly-language module used to transmit calls between two independently compiled modules. Branch islands are generated by the ILink linker to allow intrasegment calls to reach beyond 32 KB.

CFM-68K runtime architecture A 68K Mac OS runtime architecture that uses the Code Fragment Manager. Its handling of fragments and the ability to use shared libraries is analogous to that of the PowerPC runtime architecture, but it differs in a number of details because of system limitations. In particular, it uses segmented

application code addressed through a jump table. Compare **classic 68K runtime architecture, PowerPC runtime architecture.**

classic 68K runtime architecture The runtime architecture that has been used historically for 68K-based Macintosh computers. Its defining characteristics are the A5 world, segmented applications addressed through the jump table, and the application heap for dynamic storage allocation. Compare **CFM-68K runtime architecture, PowerPC runtime architecture.**

closure The set of connections for a root fragment and all the import libraries required for its execution. See also **root fragment.**

closure ID A unique value assigned by the Code Fragment Manager to each active closure.

code fragment See **fragment.**

Code Fragment Manager (CFM) The part of the Mac OS that loads fragments into memory and prepares them for execution. There are separate internal components to manage PowerPC code and CFM-68K code, but the APIs to the Code Fragment Manager are identical for each type of code. In general, the context determines which version of the Code Fragment Manager is being referred to.

code fragment resource In CFM-based runtime architectures, a resource of type `'cfrg'` with ID 0. The code fragment resource contains information used by the Code Fragment Manager to identify and prepare fragments.

code resource A resource created by the linker that contains the program's code. Code resources can be of many types—most commonly `'CODE'`, `'MPST'`, or `'DRVR'`.

code section A part of a fragment that holds executable code. The code must be position independent and read-only. A fragment may contain multiple code sections.

connection An incarnation of a fragment within a process. A fragment may have several unique connections, each local to a particular process.

connection ID A unique value assigned by the Code Fragment Manager to each active connection.

container The physical storage area for a fragment. Containers can be a file, a section of ROM, or even a resource.

container header A data structure that contains information about a given container, such as the number of code and data sections, the number of imported symbols it requires, and so on.

data fork One of two forks of a Macintosh file. The data fork can contain text, code, or data, or it can be empty. PowerPC runtime fragments and CFM-68K shared library fragments are stored in the data fork. Compare **resource fork.**

data section The part of a fragment that contains the static data used by the code section. A fragment may contain multiple data sections.

definition stub library The import library used by the linker to resolve imports in the application (or other fragment) being linked. The definition stub library defines

the external programming interface and data format of the library. Also called *link-time library*. Compare **implementation library, stub library.**

direct data area The area of memory that can be accessed directly through the base register. The direct data area can hold data items or pointers to data items.

drop-in addition See **plug-in.**

embedding alignment The alignment of a data item within a composite data item (such as a data structure). Compare **natural alignment.**

entry point A location (offset) within a module.

epilog A sequence of code that cleans up the stack after a procedure call (restoring registers, restoring the stack pointer, and so on).

executable resource Any resource that contains executable code. See also **accelerated resource.**

export A data item or executable routine within a fragment that is made available for use by other fragments.

Extended Common Object File Format (XCOFF) An executable file format generated by some PowerPC compilers. See also **Preferred Executable Format.**

external entry point In the CFM-68K runtime architecture, the entry point to a routine when called indirectly or from another fragment. Typically this entry point allows inclusion of instructions to set up an A5 world for the called routine before entering the internal entry point. Compare **internal entry point.**

external reference A reference to a routine or variable defined in a separate compilation unit or assembly.

far model The model of the classic 68K runtime architecture that specifies 32-bit addressing for code and data. Compare **near model.**

fat application An application that contains code of two or more runtime architectures. For example, a fat application may contain both CFM-68K and PowerPC runtime code.

fat binary program Any piece of executable code (application, shared library, code resource, trap, or trap patch) that contains code of multiple runtime architectures. See also **fat application, fat library, fat resource.**

fat library A shared library that contains code of two or more runtime architectures. For example, a fat library may contain both CFM-68K and PowerPC versions of a shared library.

fat resource A resource that contains executable code for two or more runtime architectures. See also **safe fat resource.**

filename A sequence of up to 31 printing characters (excluding colons) that identifies a file.

file type The type of a file, such as `'APPL'`, `'shlb'`, or `'TEXT'`, which determines how the file is used by the Mac OS or other programs.

fragment An executable unit of code and its associated data. A fragment is produced by the linker and loaded for execution by the Code Fragment Manager.

frame pointer (FP) A pointer to the beginning of a stack frame.

hashing A method of organizing symbol information in tables that allows them to be searched for quickly.

hash word An 4-byte value that contains the length and encoded name of a symbol.

implementation library The import library that is connected at load time to the application (or other fragment) being loaded. The implementation library provides the actual executable code and data exported by the library. Also called *runtime library*. Compare **definition stub library.**

import A data item or executable routine referenced by a fragment but not contained in it. An import is identified by name to the linker, but its actual address is bound at load time by the Code Fragment Manager.

import library A shared library that is automatically loaded at runtime by the Code Fragment Manager. The library's name is bound to a client at link time. Import libraries are a subset of shared libraries. Compare **plug-in.**

initialization function A function contained in a fragment that is executed immediately after the fragment is loaded and prepared. Compare **termination routine.**

internal entry point In the CFM-68K runtime architecture, the entry point to a routine when accessed through a direct call. The internal entry point skips any A5 switching and simply enters the beginning of the actual routine. Compare **external entry point.**

intersegment reference In 68K-based runtime architectures, a reference to a routine in another segment.

intrasegment reference In 68K-based runtime architectures, a reference to a routine in the same segment.

jump table In 68K-based runtime architectures, a table that contains one entry for every externally referenced routine in an application or MPW tool and provides the means by which segments are loaded and unloaded.

leaf procedure A routine that calls no other routines.

linkage area The area in the PowerPC stack that holds the calling routine's RTOC value and the saved values of the Condition Register and the Link Register. Compare **parameter area.**

link-time library See **definition stub library.**

main segment In 68K-based runtime architectures, the segment that contains the main entry point.

main symbol For applications, the main routine or main entry point. Shared libraries do not require a main symbol.

Mixed Mode Manager The part of the Mac OS that allows code with different calling conventions to call each other. For

example, the Mixed Mode Manager makes it possible for PowerPC code to call emulated classic 68K code.

module A contiguous region of memory that contains code or static data; the smallest unit of memory that is included or removed by the linker. See also **segment.**

natural alignment The alignment of a data type when allocated in memory or assigned a memory address. Compare **embedding alignment.**

near model The default model of the classic 68K runtime architecture, which specifies 16-bit addressing for code and data. Compare **far model.**

noncode resource A resource containing the data structures on which the program operates, for example, `'WIND'`, `'DLOG'`, `'DITL'`, or `'SIZE'` resources. You use the resource compiler Rez or a resource editor to create noncode resources.

parameter area The area in the PowerPC stack that holds the parameters for any routines called by a given routine. Compare **linkage area.**

PEF See **Preferred Executable Format.**

PEF container An addressable entity that contains PEF information.

plug-in A shared library that must be explicitly prepared by the client application before use. Plug-ins typically contain code and data that extend the capabilities of an application. Also called an *application extension* or a *drop-in addition*. Compare **import library.**

PowerPC application An application that contains code only for a PowerPC microprocessor. Compare **68K application**, **fat application.**

PowerPC-based Macintosh computer Any computer containing a PowerPC CPU that runs the Mac OS. Compare **68K-based Macintosh computer.**

PowerPC microprocessor Any member of the family of PowerPC microprocessors. Members of the PowerPC family include the MPC601, 603, and 604 CPUs.

PowerPC runtime architecture The runtime architecture for Mac OS–based computers using the PowerPC microprocessor. Its characteristics include storage of code and data in contiguous fragments, the absence of an A5 world, and the ability to use shared libraries. Compare **CFM-68K runtime architecture, classic 68K runtime architecture.**

Preferred Executable Format (PEF) The format of executable files used for PowerPC applications and shared libraries. It is also used for CFM-68K runtime import libraries that have been flattened. CFM-68K runtime applications are stored in a combination of PEF containers and `'CODE'` resources. See also **Extended Common Object File Format.**

preparation A general term in CFM-based runtime architectures to describe the actions of the Code Fragment Manager prior to executing a fragment. These actions include identifying imports, bringing fragments into memory, and resolving imports.

private connection A connection that cannot be shared between closures. A fragment can have multiple private

connections within a process, all serving the same client. Private connections are not visible as import libraries

private resource Any executable resource whose behavior is defined by your application (or other kind of software) alone.

process A prepared application and its associated fragments (including plug-ins). A process holds connections and closures.

program An executable unit that contains executable code (stored in either the data fork or resource fork) and noncode resources (stored in the resource fork). See also **code resource, noncode resource.**

prolog A sequence of code that prepares the stack for a procedure call (by saving registers, adjusting the stack, and so on).

Red Zone On PowerPC-based computers, the area of memory immediately above the address pointed to by the stack pointer. The Red Zone is reserved for temporary use by a routine's prolog and as an area to store a leaf procedure's nonvolatile registers.

reference The location within one module that contains the address of another module or entry point.

reference count For each prepared fragment, a value indicating the number of closures that contain the fragment.

relocation The process of replacing references to symbols with actual addresses during fragment preparation.

relocation block A 2-byte portion of relocation instruction information. A relocation instruction can span one or more relocation blocks.

resource A data structure used to store a program's data or code. This structure is declared and defined using the Rez language. Resources used to store code are built by the linker; resources used to store data are built by a resource compiler.

resource attributes Values associated with a resource that determine where and when the resource is loaded in memory, whether it can be changed, and whether it can be purged.

resource fork One of two forks of a Macintosh file. It can contain code resources or noncode resources, or it can be empty. 68K-based runtime applications store their code in the resource fork. Compare **data fork.**

resource specification The information used to identify a resource: the resource name, the resource type, and the values of its attributes.

root fragment The initial fragment in the preparation process when the Code Fragment Manager prepares a fragment and its imports.

routine descriptor A data structure used by the Mixed Mode Manager to execute a routine. A routine descriptor contains information about the routine being called such as its architecture and calling conventions. Defined by the `RoutineRecord` data type.

runtime architecture A set of basic rules that define how software operates. It dictates how code and data are addressed, the form of generated code, how applications are handled, and how to enable system calls.

The runtime architecture defines the core of the runtime environment. Compare **runtime environment.**

runtime environment The execution environment provided by the Process Manager and other system software services. The runtime environment dictates how executable code is loaded into memory, where data is stored, and how routines call other routines and system software routines. Compare **runtime architecture.**

runtime library See **implementation library.**

safe fat resource A fat resource that contains extra classic 68K code at its entry point to check for the presence of the Code Fragment Manager. This guards against calling the Mixed Mode trap when the Mixed Mode Manager is not present. See also **fat resource.**

section A storage unit in a PEF container that contains object code or data. PEF containers usually contain multiple sections.

section header A data structure in a PEF container that contains information (size, alignment, and so on) about the sections stored within it.

segment A named collection of modules in 68K-based runtime programs.

segment header A collection of fields that provides information about a segment. In the classic 68K near model architecture, it describes the location of the jump table and the number of jump table entries.

segment relocation information Part of a segment header used to store information that allows the relocation of intrasegment references for programs compiled and linked using the `-model far` option.

shadow library A small stub library that can load a larger import library on demand.

shared library A fragment that exports functions and global variables to other fragments. A shared library is not included with the application code at link time but is linked in dynamically at runtime. A shared library is stored in a file of type `'shlb'`. There are two types of shared libraries: import libraries and plug-ins.

shared library transition vector An 8-byte transition vector in the CFM-68K runtime environment. Its two fields contain the address of a function and the value to be placed in A5 when the function executes. A transition vector for a flattened shared library is identical to the PowerPC transition vector. See also **application transition vector, transition vector.**

stack An area of memory in the application partition that is used for temporary storage of data during the operation of that application or other software.

stack frame The area of the stack used by a routine for its parameters, return address, local variables, and temporary storage.

stack pointer (SP) A pointer to the top of the stack.

stand-alone code A type of program used to supplement the standard features provided by the Mac OS , to execute startup functions, or to control peripherals. This term generally refers to classic 68K programs.

static library A library whose code is included in the application at link time.

stub library A library that contains symbol definitions but no code. See also **definition stub library.**

switch frame A stack frame, created by the Mixed Mode Manager during a mode switch, that contains information about the routine to be executed, the state of various registers, and the address of the previous frame.

termination routine A function contained in a fragment that is executed just before the fragment is unloaded. Compare **initialization function.**

transition vector In the PowerPC runtime architecture, an 8-byte data structure that describes the entry point and base register address of a routine. In the CFM-68K runtime environment, a structure that contains the entry point address of a function and the value to be placed in the A5 register when the function executes. A CFM-68K transition vector may be 12 bytes long or 8 bytes long depending on whether it is created for an application or a shared library. See also **application transition vector**, **shared library transition vector.**

universal procedure pointer A generalized procedure pointer that can be either a 68K procedure pointer or the address of a routine descriptor.

update library A shared library that contains additions or changes to an existing import library.

weak library A shared library that does not need to be present at runtime for the client application to run. Sometimes called a *soft library.*

weak symbol A symbol that does not need to be present in any of the client application's import libraries at runtime. Also known as a *weak import* or *soft import.*

XCOFF See **Extended Common Object File Format.**

Index

Numerals

A, B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X, Y, Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe Illustrator™ and Adobe Photoshop™.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER
Jun Suzuki

DEVELOPMENTAL EDITORS
Laurel Rezeau and Jeanne Woodward

ILLUSTRATORS
Sandee Karr and Deb Dennis

PRODUCTION EDITOR
Alex Solinski

Special thanks to Alan Lillich, Jeff Cobb, Dave Peterson, Quinn, Erik Eidt, Cheryl Ewy, Fred Forsman, Scott Fraser, Peri Frantz, Nick Kledzik, and Karen Wenzel.

Acknowledgments to all the Core Tools, Runtime, and DTS reviewers