NOTE: This Technical Note has been retired. Please see the Technical Notes page for current documentation.

Technical Note TN1083

Weak-Linking to a Code Fragment Manager-based Shared Library

CONTENTS

Many pieces of the Mac OS are now distributed as shared libraries, including (but not limited to) QuickTime, QuickDraw 3D, Apple Game Sprockets, and the Color Picker. In many cases, developers want to use a particular technology without always having to rely on its being present. While the Mac OS has had multiple technologies for shared libraries in the past, Apple is now standardizing on the Code Fragment Manager (CFM). This Note describes how to weak-link to a CFM-based shared library, and how to check to make sure that the library is available at runtime.

This Note is aimed at all Macintosh developers who are using shared libraries.

Updated: [Apr 07 1997]

The Old Way: Installing Gestalt Selectors and Calls into Trap Tables

In the classic 68K model, all system functions are dispatched through the trap table. New system software features simply patch themselves into previously unused sections of the table. In order to expose these features to applications, each piece of the system software installs a Gestalt selector. Applications that want to check on a particular feature make a Gestalt call to ensure that its selector is installed. For example, code to check for QuickTime looks like this:

```
theErr = Gestalt (gestaltQuickTime, &result);
if (theErr != noErr)
        return false;
else
        return true;
```

Ultimately, there are two tests that you can perform: (1) whether a manager is available, and (2) whether specific features within that manager are available, e.g., is 32-bit QuickDraw available? In the second case, the approach was to use Gestalt bits to identify "chunks" of functionality.

Problems With the Old Model

Many problems arose with the old model. One was that there were a limited number of locations in the Trap Table and eventually we would run out. To work around this, multiple system calls were often placed into a single trap, with a separate selector parameter to choose the correct routine. As a consequence of this workaround, it became difficult to patch selector-based traps successfully, limiting the ability to fix bugs in the system software.

Another problem is that most patches have to be loaded at extension time, which means that they must go into the system heap. These items are loaded even if they aren't being used by any application.

If Virtual Memory is enabled, it is not allowed to swap out blocks of memory on the System Heap, because it might swap out drivers or other crucial pieces of code. As a consequence, a large system heap severely limits the amount of memory available to VM. This means that the system may spend extra time swapping code or data in and out of memory, reducing performance.

Still another problem was the difficulty of implementing and accessing global variables in pieces of code other than applications.

Given all of these limitations with the old model, it became apparent that a new model was needed.

Back to top

The New Way: Shared Libraries

The new model uses Code Fragments -- a form of shared library. All forms of code are consistently packaged as Code Fragments, with full access to global variables. When Virtual Memory is used on PowerMacs, fragments are file-mapped into memory and are marked as read-only. This results in more efficient memory usage and some protection against programming errors.

If a fragment needs access to system software features, it imports them from another library. Most pieces of the system software are imported from a library named InterfaceLib, while others (e.g., QuickTime, QuickDraw 3D) place all of their functionality into their own library.

Strong-Linking

By default, all import libraries are strong-linked, which means that the application requires them to be available in order for the application to launch. When a CFM-based application is launched, it attempts to import all the libraries. If it cannot prepare a strong-linked library, the Finder will display an error message. For example:

```
The Application "MoofWars" could not be opened because "DrawSprocketLib" could not be found.
```

> **Note:**
> This applies to any CFM shared library, not simply an application.

Weak-Linking

An application can specify a library as being weak-linked. If a library is weak-linked, the application will launch even if that library is not available or can't be prepared due to memory constraints. Once a weak library is missing, it stays missing until the application is relaunched. Weak-linking is not a substitute for call-time binding of imports. CFM only does prepare-time binding of imports. In order to use a weak-linked library, we now need to determine if that library was prepared.

An entire imported library may be marked weak, meaning that it can be entirely missing. Individual imported symbols may also be marked weak, which means they can individually be missing. Marking a library weak implicitly marks all symbols from that library as weak.

> **Note:**
> The converse is not literally true from CFM's implementation. A strong library must be available even if all symbols imported from it are weak. A "smart" linker would "realize" that all the imported symbols were weak and mark the entire library as weak.

There are some "implementation details" of CFM that can make its behavior appear inconsistent. Currently, almost any failure when identifying import libraries will let weak libraries pass. This may change in the future, so that weak libraries are only allowed to literally be missing. Once the library is successfully found, all other failures are fatal. This will not change: for example, if there is insufficient application heap space to allocate the library's data section, the prepare will fail. Because implementation details may change in the future, you should refer to the latest draft of *Macintosh Runtime Architectures* , cited at the end of this Note.

The development environment is responsible for providing an interface to weak-linking. For example, CodeWarrior only permits the entire library to be weak-linked. This is presented as a pop-up menu in the Project window for each source library. In MPW, by contrast, you can either mark an entire library or individual symbols. Consult the documentation of your development environment for specific information on to how mark libraries or individual symbols; the reason for this is that the features that those tools provide are always changing -- e.g., MPW is currently being revised to make it easier to weak-link individual symbols.

Back to top

Checking the Library Imports

A code fragment contains an address for each symbol that it imports. When a fragment is prepared, CFM determines the address and fills in the pointers. In C, these pointers are exactly what you get when you look at the address of the imported routine or variable. When an imported symbol is not resolved, its address will be set to kUnresolvedCFragSymbolAddress (which is just a long-winded way of saying zero). The most common method of testing whether a shared library was successfully prepared is to pick one symbol in the library and compare it to kUnresolvedCFragSymbolAddress. For example, if you weak-linked to QuickDraw 3D, you would use the following code to make sure it was successfully prepared:

```
#include <QD3D.h>
#include <CodeFragments.h>

Boolean HasQuickDraw3D (void)
{
    return ( (Ptr) Q3Initialize != (Ptr) kUnresolvedCFragSymbolAddress);
}
```

The presence of this symbol does not necessarily mean that all other symbols are also available. For example, if you are using ColorSync 1.0 and weak-link individual symbols from ColorSync 2.0, then you would separately test for the 2.0 symbols.

Another method for testing for a library is to call `GetSharedLibrary` using the `kFindCFrag` option. In order to do so, you need to obtain the name of the import library from the library's 'cfrg' resource.

A Correction to the Code in *3D Graphics Programming With QuickDraw 3D*

The code provided above supersedes the code provided in *3D Graphics Programming With QuickDraw 3D* . That code casts the function pointer to a Boolean, clearing the top 24 bits before testing the address. The following disassembly shows this:

```
Boolean BuggyHasQuickDraw3D (void)
{
return((Boolean) Q3Initialize != kUnresolvedCFragSymbolAddress);
}

00000084: 80C20000  lwz      r6,Q3Initialize(RTOC)
00000088: 54C6063E  clrlwi   r6,r6,24
0000008C: 28060000  cmplwi   r6,$0000
```

Back to top

Gestalt is Not Enough!

A few pieces of system software also include extensions that install Gestalt calls. For example, QuickTime needs to be able to run under both CFM and the classic 68K model, so it still patches the trap table and installs a Gestalt function. QuickDraw 3D actually includes extension code that does nothing more than install a Gestalt selector.

If you are writing an PowerPC native or CFM-68K application that imports from one of these pieces of system software, it is possible for the Gestalt selector to be installed, but for the library to be unavailable to your application when it launches. So, if you weak-link to the library, you have to check both the Gestalt selector and the library imports; otherwise, your application will crash the first time you call a routine in the library:

```
Boolean HasQuickTime (void)
{
    OSErr theErr = noErr;
    long result;

    theErr = Gestalt (gestaltQuickTime, &result);
    if (theErr != noErr)
        return false;

    theErr = Gestalt (gestaltQuickTimeFeatures, &result);
    if (theErr != noErr)
        return false;

    if (!(result & (1 << gestaltPPCQuickTimeLibPresent)))
        return false;

    // This is only required if we weak link, as the entry points for
    // the QuickTime PowerPlug might not have been resolved.

#if GENERATINGCFM
    return ((Ptr) EnterMovies != (Ptr)kUnresolvedCFragSymbolAddress);
#else
    return true;
#endif
}
```

Note that the Code Fragment Manager now also exists in the CFM68K runtime, so it is not sufficient merely to test for PowerPC. The header file ConditionalMacros.h sets up a number of conditionals, including GENERATINGCFM, which is set to true when compiling under PowerPC or CFM68K. Wrapping your test for weak-linked symbols in #if GENERATINGCFM is the recommended way to conditionalize the test.

Back to top

Summary

Apple has now standardized on the Code Fragment Manager, which offers a consistent interface for all code in the Mac OS. The Code Fragment Manager has also moved back to the 68K. Weak-linking allows a particular fragment to not be dependent on another fragment. You must test to make sure that weak-linked fragments are available before using them.

Back to top

References

Inside Macintosh: PowerPC System Software

3D Graphics Programming With QuickDraw 3D, p. 1-16

A draft of *Macintosh Runtime Architectures* is available in E.T.O.#21

Back to top

Downloadables

Acrobat version of this Note (K). Download

Back to top

Technical Notes by API | Date | Number | Technology | Title
Developer Documentation | Technical Q&As | Development Kits | Sample Code