TECHNOTE:

Calling CFM Code From Classic 68K Code, or There and Back Again, A Mixed Mode Magic Adventure[1]

By George Warner
geowar@apple.com
Apple Developer Technical Support (DTS)

There are specific instances when you must call **Code Fragment Manager** (CFM) code from classic 68K code — for example, if your application cannot be converted to CFM, but you want to be able to use CFM libraries. Or, you want to add plug-in support to an existing classic 68K application without having to convert it to CFM68K. In addition, you may want to use CFM68K to develop an application to run on both 68K and PowerPC computers and use a single FAT library for both environments.

Another instance would be developing for OpenDoc, which requires shared library support. Prior to this Technote, only CFM applications could take advantage of OpenDoc. This Technote explains how to add library support to classic code.

1 with apologies to Bilbo Baggins, from J.R.R. Tolkien's Lord of the Rings

Calling CFM From Classic Code

The basic steps for calling CFM from classic code are as follows:

1. Determine the address of the CFM routines you want to call.
2. Create a routine descriptor for the CFM routine.
3. Call the routine descriptor.
4. Clean-up after yourself.

Determining the Address of the CFM Routines

The most common way to determine the address of a CFM routine is to use `FindSymbol` against a shared library.

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

The first parameter is a connection ID to a fragment. This can be obtained from a call to `GetSharedLibrary`, `GetDiskFragment` or `GetMemoryFragment`. The second parameter is the name of the symbol, in this case the name of the routine we want to call. The address of the symbol is returned in the third parameter and the last parameter returns the symbols class.

Creating a Routine Descriptor

In CFM code, you normally use the NewRoutineDescriptor routine to create routine descriptors. The non-CFM version of this is the NewRoutineDescriptorTrap() routine. See the notes on using New[Fat]RoutineDescriptor[Trap] at the end of this Technote.

```
pascal UniversalProcPtr NewRoutineDescriptorTrap(
 ProcPtr theProc,
 ProcInfoType theProcInfo,
 ISAType theISA);
```

To create a routine descriptor, you need the address, the procedure information and the architecture of the routine being described. The address was obtained in the first step. The procedure information is based on the calling conventions for the parameters passed to and returned from the routine. The last parameter is the architecture of the routine being called. In the headers, this is defined as the ISAType, which is a combination of two separate 4 bit values, the **Instruction Set Architecture** (ISA) and **Runtime Architecture** (RTA).

Determining the Procedure Information

The Mixed Mode Manager supports many calling conventions. The most common are stack based for Pascal and C, register based for operating system traps and register dispatched for selector based Toolbox traps. The easiest way to define the procedure information is via a enum: for example, if your routine was defined like this:

```
pascal Ptr Get_Message(short pResID, short pIndex);
```

Its procedure information would then be defined like this:

```
enum {
    kGet_MessageProcInfo = kPascalStackBased
  | RESULT_SIZE(SIZE_CODE(sizeof(Ptr)))
      | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(short)))
  | STACK_ROUTINE_PARAMETER(2, SIZE_CODE(sizeof(short)))
};
```

Note: The RESULT_SIZE, SIZE_CODE & STACK_ROUTINE_PARAMETER macros are defined in MixedMode.h. ProcInfo is documented in Chapter Two, Inside Macintosh: PowerPC System Software.

Determining the Architecture

Everyone seems to be tempted to use `GetCurrentArchitecture` here; JUST SAY NO! `GetCurrentArchitecture` is a macro that returns the architecture of the currently compiling code, in our case, classic. What we want is the architecture of the CFM code that we are calling from our classic code. Remember: the architecture includes the ISA and the RTA. The correct thing to do is to call Gestalt to find out what kind of CPU the code is running on (68K or PPC) and use this to create the routine descriptor with the correct architecture:

```
static pascal OSErr GetSystemArchitecture(OSType *archType)
{
  // static so we only Gestalt once
  static long sSysArchitecture = 0;
  OSErr tOSErr = noErr;
```

```
  // assume wild architecture
  *archType = kAnyCFragArch;

  // If we don't know the system architecture yet…
  // Ask Gestalt what kind of machine we are running on.
  if (sSysArchitecture == 0)
   tOSErr = Gestalt(gestaltSysArchitecture,
       &sSysArchitecture);

  if (tOSErr == noErr) // if no errors
  {
   if (sSysArchitecture == gestalt68k) // 68k?
    *archType = kMotorola68KArch;
   else if (sSysArchitecture == gestaltPowerPC) // PPC?
    *archType = kPowerPCArch;
   else
    tOSErr = gestaltUnknownErr;
    // who knows what might be next?
  }
 return tOSErr;
}
```

Note: Don't confuse the OSType architecture used to specify fragment architecture with the SInt8 ISA/RTA architecture used to specify routine descriptors. This routine determines the OSType architecture. I did it this way because I use it to open the connection to my shared library, which requires the OSType. When I create my routine descriptors, I use this value to conditionally execute the `NewRoutineDescriptorTrap` routine with the correct ISA/RTA type parameter:

```
 static OSType sArchType = kAnyCFragArch;
 ISAType tISAType;
 if (sArchType == kAnyCFragArch) // if architecture is still undefined…
 {
  // & determine current atchitecture.
  sOSErr = GetSystemArchitecture(&sArchType);
```

```
 if (sOSErr != noErr)
  return sOSErr;
}
if (sArchType == kMotorola68KArch) // …for CFM68K
tISAType = kM68kISA | kCFM68kRTA;
else if (sArchType == kPowerPCArch)  // …for PPC CFM
tISAType = kM68kISA | kCFM68kRTA;
else
 sOSErr = gestaltUnknownErr; // who knows what might be next?
if (sOSErr == noErr)
 myUPP = NewRoutineDescriptorTrap((ProcPtr) * pSymAddr,
   pProcInfo,tISAType);
return sOSErr;
```

Calling the Routine Descriptor

From CFM code you normally use `CallUniversalProc` to call the routine associated with a routine descriptor. (A Universal Procedure Pointer (UPP) is a pointer to a routine descriptor.) However, `CallUniversalProc` is only implemented in shared libraries. For compatability reasons, in classic code UPP's can be treated like ProcPtr's, i.e., they are pointers to executable code. This is because the first two bytes of a routine descriptor is the 68K mixed mode magic ATrap. When we jump here from 68K code, the trap is executed and the Mixed Mode Manager takes over, setting up passed parameters in registers or on the stack, based on the ProcInfo in the routine descriptor: switching the architecture and then jumping to the CFM code. This is how you would call your routine:

```
// define a ProcPtr of our type
typedef pascal Ptr (*Get_Message_ProcPtr)(short pResID,
  short pIndex);

// call it.
```

```
myPtr = ((Get_Message_ProcPtr)myUPP)(128,3);
```

Cleaning Up After Yourself

`DisposeRoutineDescriptorTrap` is used to release the memory allocated for routine descriptors by the `NewRoutineDescriptorTrap`.

Shortcuts, Detours & Dead-Ends

You may be tempted to use the macro `BUILD_ROUTINE_DESCRIPTOR`, so that you can build your routine descriptors staticly. Unfortunately, this macro expands to include the macro `GetCurrentArchitecture` whose problem was described in the section above. Another problem with this approach is that the `ProcPtr` passed to the macro is expected to be a constant at compile-time. One solution to both of these problems is to build your routine descriptors in your CFM library and export them. This way the `GetCurrentArchitecture` macro returns the correct architecture for the library and the `ProcPtr` is a compile-time constant. And since these routine descriptors are staticly allocated at compile time, you don't have to worry about disposing them: their memory is released when the library is unloaded. Unfortunately, this only works if you have source to the library you want to connect to.

Using BUILD_ROUTINE_DESCRIPTOR to dynamically initialize a routine descriptor is not a good idea. From the classic 68K perspective, the routine descriptor is code being assembled out of data. This can cause problems `due to the split caches on 68040 CPUs and some 68K emulator optimizations on PowerPCs`. You're trying to execute data but instead are executing old values from the instruction cache. Using NewRoutineDescriptorTrap insures that the instruction cache is flushed for the executable range of the routine descriptor - two bytes.

In order to make the connection between classic code and the CFM code as transparent as possible, I like to put all my CFM glue code in its own separate file and use the same API in it as defined for my library (usually by using the library's header file). Each entry point into the library has its own glue routine that declares a static UPP variable initialized to `kUnresolvedSymbolAddress`. By checking for this initial value, the routine knows when it needs to look up its address in the library and create a routined descriptor. Here's the glue code for the library:

Source Code for CFM Library Glue

```
#include <CodeFragments.h>
#include "DemoLib.h"


// Private function prototypes

static OSErr Find_Symbol(Ptr* pSymAddr,
```

```
        Str255 pSymName,
        ProcInfoType pProcInfo);

static pascal OSErr GetSystemArchitecture(OSType *archType);

// Private functions

static pascal OSErr GetSystemArchitecture(OSType *archType)
{
 static long sSysArchitecture = 0; // static so we only Gestalt once.
 OSErr tOSErr = noErr;

 *archType = kAnyCFragArch;   // assume wild architecture

 // If we don't know the system architecture yet…
 if (sSysArchitecture == 0)
  // …Ask Gestalt what kind of machine we are running on.
  tOSErr = Gestalt(gestaltSysArchitecture, &sSysArchitecture);

 if (tOSErr == noErr) // if no errors
 {
  if (sSysArchitecture == gestalt68k)   // 68k?
   *archType = kMotorola68KCFragArch;
  else if (sSysArchitecture == gestaltPowerPC) // PPC?
   *archType = kPowerPCCFragArch;
  else
   tOSErr = gestaltUnknownErr;  // who knows what might be next?
 }
 return tOSErr;
}

static OSErr Find_Symbol(Ptr* pSymAddr,
        Str255 pSymName,
        ProcInfoType pProcInfo)
{
 static ConnectionID sCID = 0;
 static OSType sArchType = kAnyCFragArch;
 static OSErr sOSErr = noErr;

 Str255 errMessage;
 Ptr mainAddr;
 SymClass symClass;
   ISAType     tISAType;

 if (sArchType == kAnyCFragArch)  // if architecture is undefined…
 {
  sCID = 0;     // …force (re)connect to library
  sOSErr = GetSystemArchitecture(&sArchType); // determine architecture
  if (sOSErr != noErr)
   return sOSErr; // OOPS!
 }

 if (sArchType == kMotorola68KArch) // …for CFM68K
   tISAType = kM68kISA | kCFM68kRTA;
 else if (sArchType == kPowerPCArch)  // …for PPC CFM
   tISAType = kM68kISA | kCFM68kRTA;
 else
  sOSErr = gestaltUnknownErr; // who knows what might be next?

 if (sCID == 0) // If we haven't connected to the library yet…
 {
  // NOTE: The library name is hard coded here.
  // I try to isolate the glue code, one file per library.
  // I have had developers pass in the Library name to allow
  // plug-in type support. Additional code has to be added to
  // each entry points glue routine to support multiple or
  // switching connection IDs.
```

```
  sOSErr = GetSharedLibrary("\pDemoLibrary", sArchType, kLoadCFrag,
     &sCID, &mainAddr, errMessage);
  if (sOSErr != noErr)
   return sOSErr; // OOPS!
 }

 // If we haven't looked up this symbol yet…
 if ((Ptr) *pSymAddr == (Ptr) kUnresolvedCFragSymbolAddress)
 {
  // …look it up now
  sOSErr = FindSymbol(sCID,pSymName,pSymAddr,&symClass);
  if (sOSErr != noErr) // in case of error…
   // …clear the procedure pointer
   *(Ptr*) &pSymAddr = (Ptr) kUnresolvedSymbolAddress;
#if !GENERATINGCFM // if this is classic 68k code…
   *pSymAddr = (Ptr)NewRoutineDescriptorTrap((ProcPtr) *pSymAddr,
         pProcInfo, tISAType);  // …create a routine descriptor…
#endif
 }
 return sOSErr;
}

/* Public functions & globals */

pascal void Do_Demo(void)
{
 static Do_DemoProcPtr sDo_DemoProcPtr = kUnresolvedSymbolAddress;

 // if this symbol has not been setup yet…
 if ((Ptr) sDo_DemoProcPtr == (Ptr) kUnresolvedSymbolAddress)
  Find_Symbol((Ptr*) &sDo_DemoProcPtr,"\pDo_Demo",kDo_DemoProcInfo);
 if ((Ptr) sDo_DemoProcPtr != (Ptr) kUnresolvedSymbolAddress)
  sDo_DemoProcPtr();
}



pascal void Set_DemoValue(long pLong)
{
 static Set_DemoValueProcPtr sSet_DemoValueProcPtr =
  kUnresolvedSymbolAddress;

 // if this symbol has not been setup yet…
 if ((Ptr) sSet_DemoValueProcPtr == (Ptr) kUnresolvedSymbolAddress)
  Find_Symbol((Ptr*) &sSet_DemoValueProcPtr,
   "\pSet_DemoValue", kSet_DemoValueProcInfo);
 if ((Ptr) sSet_DemoValueProcPtr != (Ptr) kUnresolvedSymbolAddress)
  sSet_DemoValueProcPtr(pLong);
}

pascal long Get_DemoValue(void)
{
 static Get_DemoValueProcPtr sGet_DemoValueProcPtr =
  kUnresolvedSymbolAddress;

 // if this symbol has not been setup yet…
 if ((Ptr) sGet_DemoValueProcPtr == (Ptr) kUnresolvedSymbolAddress)
  Find_Symbol((Ptr*) &sGet_DemoValueProcPtr,
   "\pGet_DemoValue",kGet_DemoValueProcInfo);
 if ((Ptr) sGet_DemoValueProcPtr != (Ptr) kUnresolvedSymbolAddress)
  return sGet_DemoValueProcPtr();
 else
  return 0L;
}

pascal Ptr Get_DemoString(void)
{
```

```
 static Get_DemoStringProcPtr sGet_DemoStringProcPtr =
  kUnresolvedSymbolAddress;

 // if this symbol has not been setup yet…
 if ((Ptr) sGet_DemoStringProcPtr == (Ptr) kUnresolvedSymbolAddress)
  Find_Symbol((Ptr*) &sGet_DemoStringProcPtr,
   "\pGet_DemoString",kGet_DemoStringProcInfo);
 if ((Ptr) sGet_DemoStringProcPtr != (Ptr) kUnresolvedSymbolAddress)
  return sGet_DemoStringProcPtr();
 else
  return 0L;
}
```

Note: The above routines will silently do nothing if their Find_Symbol call fails. Routines that do this sort of load/resolve on the fly should always have a means to bail out in case there are any errors. For example, return OSErr, use some kind of exception mechanism, etc. At the least, have Find_Symbol put up a fatal alert. This is left as an exercise for the programmer.

Notes on Using the New[Fat]RoutineDescriptor[Trap]

When calling NewRoutineDescriptor from classic 68K code, there are two possible intentions. The first is source compatibility with code ported to CFM (either Power PC or 68K CFM). When the code is compiled for CFM, the functions create routine descriptors that can be used by the mixed mode manager operating on that machine. When the code is compiled for classic 68K, these functions do nothing so that the code will run on Macintoshes that do not have a Mixed mode manager. The dual nature of these functions is achieved by turning the CFM calls into "no-op" macros for classic 68K: You can put "NewRoutineDescriptor" in your source, compile it for any architecture, and it will run correctly on the intended platform. All without source changes and/or conditional source.

The other intention is for code that "knows" that it is executing as classic 68K and is specifically trying to call code of another architecture using mixed mode. Since the routines were designed with classic <-> CFM source compatibility in mind, this second case is treated special. For classic 68k code to create routines descriptors for use by mixed mode, it must call the "Trap" versions of the routines (NewRoutineDescriptorTrap). These versions are only available to classic 68K callers: rigging the interfaces to allow calling them from CFM code will result in runtime failure because no shared library implements or exports these functions.

This almost appears seamless until you consider "fat" routine descriptors and the advent of CFM-68K. What does "fat" mean? CFM-68K is not emulated on Power PC and Power PC is not emulated on CFM-68K. It makes no sense to create a routine descriptor having both a CFM-68K routine and a Power PC native routine pointer. Therefore "fat" is defined to be a mix of classic and CFM for the hardware's native instruction set: on Power PC fat is classic and Power PC native, on a 68k machine with CFM-68K installed fat is classic and CFM-68K.

By definition fat routine descriptors are only constructed by code that is aware of the architecture it is executing as and that another architecture exists. Source compatibility between code intended as pure classic and pure CFM is not an issue

and so NewFatRoutineDescriptor is not available when building pure classic code.

NewFatRoutineDescriptorTrap is available to classic code on both Power PC and CFM-68K. The classic code can use the code fragment manager routine "FindSymbol" to obtain the address of a routine in a shared library and then construct a routine descriptor with both the CFM routine and classic routine.

About Mixed Mode and Routine Descriptors

In the beginning (1984), there was the classic Macintosh programming model, based on the Motorola 680x0 processor and code segments. Then in 1991, the PowerPC processor was introduced. There was concern about compatibility with existing 68K applications (including the Finder), and the first step in addressing this concern was writing a 68LC040 emulator which allowed 68K code to run unmodified in the new environment. As part of this effort, a method had to be devised to switch between the native PPC and the emulated 68K modes — thus, the Mixed Mode Manager was born.

The Mixed Mode Manager is system software that manages mode switches between code in different **instruction set architectures** (ISA's). An ISA is the set of instructions recognized by a particular processor or family of processors. You indicate the ISA of a particular routine by creating a **routine descriptor** for that routine.

Note: For more information about the Mixed Mode Manager, read its chapter in *Inside Macintosh: PowerPC System Software*. The documentation also applies to CFM68K — just consider "native" code to be either PowerPC or CFM68K.

Code Fragment Manager

CFM was developed initially for PowerPC-based Macintosh computers to prepare code fragments for execution. A fragment is a block of executable code and its associated data. On PowerPC-based Macintosh computers, native programs, applications, libraries, and standalone code are packaged as fragments.

In 1994, CFM was ported back to 68K. The Mixed Mode Manager was again used to handle transitions between classic 68K and the CFM conventions for the CPU it is running on, i.e., on PowerPC it can handle classic to PowerPC transitions, and on 68K it can handle classic to CFM68K transitions. Classic 68K code is generally ignorant of mode switches while CFM code must be aware of them. Classic 68K code can treat a routine descriptor pointer as a classic 68K proc pointer, but CFM code cannot treat a routine descriptor as a proc pointer.

Summary

Calling CFM from classic code may be necessary for a number of reasons, particularly if you want to take advantage of both the classic and CFM libraries. It may also be the simplest and easiest method of adding plug-in support to an existing 68K or FAT application without having to port the 68K code to CFM68K.

This Technote discusses some straightforward methods you can use to call CFM code from classic code. There are problems, however, that you ought to consider when trying to build routine descriptors for C routines in a shared library.

Further Reference

- *Inside Macintosh: PowerPC System Software*
- *Fragments of Your Imagination* by Joe Zobkiw, Addison-Wesley, ISBN 0-201-48358-0

Acknowledgments

Thanks to Andy Bachorski, Alan Lillich, Dave Peterson and Bob Wambaugh