Technical Note TN1127

In Search of Missing Links

CONTENTS

They're everywhere: Missing Links! If you are writing a CFM application and want to call `FlushCodeCacheRange`, show and hide the control strip, or use the Cursor Device Manager, you will run into the same problem: missing links. These Application Programming Interfaces (API's) haven't been implemented in `InterfaceLib`. So what do you do? Well that's what this Technote is about--fixing missing links. It's the sequel to Technote 1077 on calling CFM routines from classic 68K applications. This Technote demonstrates how to write CFM glue code, which can call classic 68K A-Trap routines.

This Technote is of interest to developers attempting to use the classic 68K API's currently missing from `InterfaceLib`.

IMPORTANT:
WARNING: This is an advanced topic for use by developers who can't continue development because of these missing API's. Even as this Technote is being published, efforts are being made to identify the missing links and get them added to `InterfaceLib`.

Updated: [May 4 1998]

The Problem: Missing routines

Some classic 68K API's are missing from PowerPC & CFM-68K `InterfaceLib`. Calling these routines from CFM code will generate "Unresolved Symbol" errors when you link your program.

Back to top

The Solution: Glue Them In

This Technote provides a temporary solution by showing how to create glue code which you can include with your program. You need to write one glue routine for each missing API. Each of these routines determines the address of the 68K A-Trap and then calls it via `CallUniversalProc` or `CallOSTrapUniversalProc` with the appropriate parameters. If in the future a new `InterfaceLib` file is released that includes the missing API's, your project can be rebuilt without the glue file.

Back to top

Some (Very) Simple Examples

> **Note:**
> The following four API's are not missing from `InterfaceLib`. They were selected in order to provide simple examples.

Here is `InitWindows`, `SelectWindow`, `FrontWindow` & `CheckUpdate` from <MacWindows.h>.

```
extern pascal void InitWindows(void)
    ONEWORDINLINE(0xA912);

extern pascal void SelectWindow(WindowPtr's theWindow)
    ONEWORDINLINE(0xA91F);

extern pascal WindowPtr FrontWindow(void)
    ONEWORDINLINE(0xA924);

extern pascal Boolean CheckUpdate(EventRecord *theEvent)
    ONEWORDINLINE(0xA911);
```

For `InitWindows`, all you only need to `CallUniversalProc` with the address of the A-Trap and the Pascal `ProcInfo`, like this:

```
pascal void InitWindows(void)
// ONEWORDINLINE(0xA912);
{
    CallUniversalProc(GetToolboxTrapAddress(0xA912),
        kPascalStackBased);
}
```

That was not too hard. Just in case you don't know where the value `0xA912` came from for the `GetToolboxTrapAddress` call, it's in the `ONEWORDINLINE` macro. You could also have looked it up in <Traps.h>. OK? This time we will pass in a parameter:

```
extern pascal void SelectWindow(WindowPtr theWindow)
// ONEWORDINLINE(0xA91F);
{
    CallUniversalProc(GetToolboxTrapAddress(0xA91F),
        kPascalStackBased |
            STACK_ROUTINE_PARAMETER(1,SIZE_CODE(sizeof(WindowPtr))),
        theWindow    // the parameter
    );
}
```

Nothing complicated here--we added the `STACK_ROUTINE_PARAMETER` macro to the `procinfo` and the parameter. Let's try returning a value:

```
extern pascal WindowPtr FrontWindow(void)
// ONEWORDINLINE(0xA924);
{
    return (WindowPtr) CallUniversalProc(
        GetToolboxTrapAddress(0xA924),
        kPascalStackBased |
            RESULT_SIZE(SIZE_CODE(sizeof(WindowPtr)))
    );
}
```

This one uses the `RESULT_SIZE` macro to define the size of the result on the stack. Notice that we had to typecast the long returned by `CallUniversalProc` to a `WindowPtr`. Now, we can combine what we have learned. Here is an API that has single parameter and returns a result:

```
extern pascal Boolean CheckUpdate(EventRecord *theEvent)
// ONEWORDINLINE(0xA911);
{
    return (Boolean) CallUniversalProc(
        GetToolboxTrapAddress(0xA911),
        kPascalStackBased |
            RESULT_SIZE(SIZE_CODE(sizeof(Boolean))) |
            STACK_ROUTINE_PARAMETER(1,SIZE_CODE(sizeof(EventRecord*)))
    );
}
```

Back to top

Register-Based A-Traps:

Operating system A-Traps pass their parameters in registers. Apple recently introduced the `PBXGetVolInfo` (note the

X in *PBX*) A-Trap, which acts like PBGetVolInfo, but supports returning information about the new large volumes. Here's how you can call it from CFM:

```
// WARNING: Don't use this code - read rest of section!
//#pragma parameter __D0 PBXGetVolInfoSync(__A0)
pascal OSErr PBXGetVolInfoSync(XVolumeParamPtr paramBlock)
// TWOWORDINLINE(0x7012, 0xA060);
{
    return (OSErr) CallOSTrapUniversalProc(
        (UniversalProcPtr) GetOSTrapAddress(0xA060),
        kRegisterBased |
            RESULT_SIZE(SIZE_CODE(sizeof(OSErr))) |
            REGISTER_RESULT_LOCATION(kRegisterD0) |
            REGISTER_ROUTINE_PARAMETER(1, kRegisterD0,
                kTwoByteCode) |    // selector
            REGISTER_ROUTINE_PARAMETER(2, kRegisterA0,
                SIZE_CODE(sizeof(XVolumeParamPtr*))),
        0x0012,    // selector
        paramBlock);    // parameter(s)
}
// WARNING: Don't use this code - read rest of section!
```

The first thing to notice is that we're using CallOSTrapUniversalProc instead of CallUniversalProc. The CallOSTrapUniversalProc function executes the routine associated with the specified universal procedure pointer, following standard conventions for executing OS A-Traps. Registers A1, A2, D1, and D2 are saved before the routine is executed and restored after its completion; in addition, register A0 is saved and restored, depending on the setting of the appropriate flag bit in the trap word.

How did we determine the selector (0x0012)? If you disassemble the TWOWORDINLINE hexcodes in MacsBug, you should get this:

```
MacsBug> dh 7012 A060
    MOVEQ    #$12,D0
    _FSDispatch
```

Just in case you're confused, we are copying the API and the xWORDINLINE information directly out of Universal Interfaces. In this case the pragma parameter tells what registers are used where (OSErr result in D0 and XVolumeParamPtr parameter in A0). Just like in the previous cases, disassembling the xWORDINLINE macro gives the selector information and the A-Trap. There isn't a kD0DispatchedRegisterBased ProcInfo-calling convention so we have to treat the D0 selector like another register-based parameter.

Notice that I'm using GetOSTrapAddress instead of GetToolboxTrapAddress. This brings up another issue. Some of the OS A-Traps pass flags in bits 9 & 10 of the A-Trap. For example look at NewPtr:

```
    _NewPtr = 0xA11E
    _NewPtrSys  = 0xA51E
    _NewPtrClear= 0xA31E
    _NewPtrSysClear = 0xA71E
```

You can tell that bit 9 is the 'Sys' bit and bit 10 is the 'Clear' bit. These four A-Traps are dispatched through a single dispatcher that uses these flag bits to determine which routine to call. Guess what? The A-Trap for `PBXGetVolInfoSync` (`0xA060`) is one of these:

```
    _FSDispatch      = 0xA060
    _HFSDispatch     = 0xA260
    _FSDispatchAsync  = 0xA460
    _HFSDispatchAsync = 0xA660
```

See Inside Macintosh: Operating Systems Utilities for more information on the Trap Manager.

So how does the A-Trap dispatcher know what A-Trap invoked it? It's passed to the dispatcher in register D1. So to fix out `PBXGetVolInfoSync` glue above we need to add another parameter to tell the Mixed Mode Manager to pass the A-Trap in register D1.

Note:
The order here is critical! If CFM glue is calling a trap that has been patched with CFM code and the `ProcInfo` parameter order isn't the same, then the patch will be called with the original order of parameters. The canonical ordering is selector (if required), then D1 (the A-Trap), then the parameters in the order that they appear in the high-level language prototype.

```
pascal OSErr PBXGetVolInfoSync(XVolumeParamPtr paramBlock)
// TWOWORDINLINE(0x7012, 0xA060);
{
    return (OSErr) CallOSTrapUniversalProc (
        (UniversalProcPtr) GetOSTrapAddress(0xA060),
        kRegisterBased |
            RESULT_SIZE(SIZE_CODE(sizeof(OSErr))) |
            REGISTER_RESULT_LOCATION(kRegisterD0) |
            REGISTER_ROUTINE_PARAMETER(1, kRegisterD0,
                kTwoByteCode) |    // selector
            REGISTER_ROUTINE_PARAMETER(2, kRegisterD1,
                kTwoByteCode) |    // A-Trap
            REGISTER_ROUTINE_PARAMETER(3, kRegisterA0,
                SIZE_CODE(sizeof(XVolumeParamPtr*))),
        0x0012,        // selector
        0xA060,        // A-Trap
        paramBlock);    // parameter(s)
}
```

Back to top

Register-Dispatched A-Traps:

Register-dispatched A-Traps require a selector passed ether register D0 or D1. Here's an AppleGuide API that until recently was missing from the CFM-68K `InterfaceLib`:

```
pascal AGErr AGGeneral(AGRefNum refNum, AGEvent theEvent)
// TWOWORDINLINE(0x700D, 0xAA6E);
{
    return (AGErr) CallUniversalProc(GetToolboxTrapAddress(0xAA6E),
        kD0DispatchedPascalStackBased |
            RESULT_SIZE(SIZE_CODE(sizeof(AGErr))) |
            DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(1,
                SIZE_CODE(sizeof(refNum))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(2,
                SIZE_CODE(sizeof(theEvent))),
        0x000D,         // selector -> D0
        refNum,theEvent    // the parameters
    );
}
```

The first thing to notice about this code is the selector (`0x0012`). You should remember how to get its value from MacsBug using the values from the `TWOWORDINLINE` macro:

```
MacsBug> dh 700D AA6E
    MOVEQ    #$0D,D0
    _AGGeneral
```

This gives us a clue to determine that this API is D0-dispatched. Now, how do we determine the correct value to use for the `DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE`? Another quick trip back to MacsBug can give us a list of the API's dispatched through the `_AppleGuideDispatch` A-Trap:

```
MacsBug> api AA6E

    ° AA6E _AppleGuideDispatch
        DO.W=0001        AGOpen
        DO.W=0002        AGOpenWithSearch
        DO.W=0003        AGOpenWithSequence
        <...>
```

From this we can determine that the selector is in fact word-based; therefore, we know to use the kTwoByteCode value for the DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE.

Back to top

Stack-Dispatched A-Traps:

Stack-dispatched A-Traps require a selector passed in on the stack. While TEGetPoint isn't missing from InterfaceLib, we are going to borrow it to show one of those pitfalls.

```
// WARNING: Don't use this code - read paragraph below

static Point TEGetPoint(short offset,TEHandle hTE)
{
// THREEWORDINLINE(0x3F3C, 0x0008, 0xA83D);
// MOVE.W    #$0008,-(A7)
//        _TEDispatch
// _TEGetPoint is A83D (_TEDispatch) when A7^.W=0008
    return (Point) CallUniversalProc(
        GetToolboxTrapAddress(_TEDispatch),    // address of dispatcher
        kStackDispatchedPascalStackBased |        // proc info
            RESULT_SIZE(SIZE_CODE(sizeof(Point))) |
            DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(1,
                SIZE_CODE(sizeof(short))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(2,
                SIZE_CODE(sizeof(TEHandle))),
        0x0008,    // selector -> stack
        offset, hTE    // parameter(s)
    );
}
// WARNING: Don't use this code - read paragraph below
```

This time the selector was passed in as a word on the stack. Does everything make sense? Almost. This was a trick question to catch anyone thinking they just could skip ahead to the tough ones. In this case, the TEGetPoint returns a Point. No problem--we'll just typecast it, right? Try it. Oops, Houston, we have a problem. The compiler complains: "Error: illegal explicit conversion from 'long' to 'struct Point'." To avoid this, we have to store the results in a local variable and typecast its address as a pointer to a Point. We then de-reference this to get the correct results, like this:

```
static Point TEGetPoint(short offset,TEHandle hTE)
{
// THREEWORDINLINE(0x3F3C, 0x0008, 0xA83D);
// MOVE.W    #$0008,-(A7)
//        _TEDispatch
// _TEGetPoint is A83D (_TEDispatch) when A7^.W=0008
    long private_results = CallUniversalProc(
        GetToolboxTrapAddress(_TEDispatch),    // address of dispatcher
        kStackDispatchedPascalStackBased |    // proc info
            RESULT_SIZE(SIZE_CODE(sizeof(Point))) |
            DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(1,
                SIZE_CODE(sizeof(short))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(2,
                SIZE_CODE(sizeof(TEHandle))),
        0x0008,        // selector
        offset, hTE    // parameter(s)
    );
    return *(Point*) &private_results;
}
```

This is all well and good if the structure is four bytes in size. What would happen if it were smaller? Try this example:

```
typedef struct CryptoCharsRecord {
    char    findChar;
    char    replaceChar;
} CryptoCharsRec, *CryptoCharsPtr, **CryptoCharsHdl;
```

If this structure was to be returned from `CallUniversalProc` and stored into our long result, it would look like this (remember that the Mac OS stores the high bytes first and the low bytes last):

```
+--------------------------+
Low Mem | 0 | <-- Address of private_result
+--------------------------+
| 1 |
+--------------------------+
| 2 |    findChar
+--------------------------+
High Mem| 3 |    replaceChar
+--------------------------+
```

In this case, neither the simple typecast nor the dereferenced-pointer typecast will work. What we have to do is dereference the address of our result, plus an offset to the high word.

```
pascal CryptoCharsRec GetEncodeKey(char* pThePassword)
{
    long private_results = CallUniversalProc(
        GetToolboxTrapAddress(0xAxxx),
        kPascalStackBased |
            RESULT_SIZE(SIZE_CODE(sizeof(Boolean))) |
            STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(EventRecord*)))
        theEvent    // the parameter
    );
    return *(((CryptoCharsRec*)&private_result) + 1);
}
```

At first glance, you might think that the offset should be two, but remember that in this case it's not a byte offset, it is an offset for a two-byte structure. C knows this and internally adds two instead of one. It's really easy to miscompute this offset, so here's a macro that will compute it for you given a variable or variable type:

```
#define RESULT_OFFSET(type) \
    ((sizeof(type) == 1) ? 3 : ((sizeof(type) == 2) ? 1 : 0))
```

There's nothing complicated here. If the size of the structure is one then the offset to the high byte is three (bytes). If the size of the structure is two, then the offset to the high word is one (word); otherwise, the offset is zero. It's used like this:

```
pascal CryptoCharsRec Get_Encode_Key(char* pThePassword)
{
    long private_results = CallUniversalProc(
        GetToolboxTrapAddress(0xA911),
        kPascalStackBased |
            RESULT_SIZE(SIZE_CODE(sizeof(Boolean))) |
            STACK_ROUTINE_PARAMETER(1,
                SIZE_CODE(sizeof(EventRecord*)))
        theEvent    // the parameter
    );
    return *(((CryptoCharsRec*)&private_result) +
        RESULT_OFFSET(CryptoCharsRec));
}
```

Is this clear? So what happens if our result is larger than four bytes? If this is the case then `CallUniversalProc` can't be used to call the routine. Some of the (machine-generated) glue code I've seen includes this code to test for result sizes larger than what can be handled by `CallUniversalProc`:

```
#ifdef applec
    #if sizeof(OSErr) > 4
        #error "Result types larger than 4 bytes are not supported."
    #endif
#endif
```

Back to top

Some Real Examples:

Let's take a look at some API's actually missing from `InterfaceLib`. Here's one from the cursor device manager:

```
pascal OSErr CursorDeviceMoveTo(CursorDevicePtr ourDevice, long absX, long absY)
{
//    TWOWORDINLINE(0x7001, 0xAADB);
//    MOVEQ    #$01,D0        | 7001
//    _CursorDeviceDispatch    | AADB
    return (OSErr) CallUniversalProc(
        (UniversalProcPtr)GetToolboxTrapAddress(0xAADB),
        kD0DispatchedPascalStackBased |
        RESULT_SIZE(SIZE_CODE(sizeof(OSErr))) |
        DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kFourByteCode) |
        DISPATCHED_STACK_ROUTINE_PARAMETER(1,
            SIZE_CODE(sizeof(ourDevice))) |
        DISPATCHED_STACK_ROUTINE_PARAMETER(2,
            SIZE_CODE(sizeof(absX))) |
        DISPATCHED_STACK_ROUTINE_PARAMETER(3,
            SIZE_CODE(sizeof(absY))),
    0x00000001,                // selector
    ourDevice, absX, absY);    // parameter(s)
}
```

Now one missing from the `OSUtils.h`:

```
static pascal OSErr FlushCodeCacheRange(void *address, unsigned long count)
{
//  TWOWORDINLINE(0x7009, 0xA098);
//  MOVEQ  #$09,D0| 7009
//  _HWPriv  | A098
    return (OSErr) CallOSTrapUniversalProc(
        GetOSTrapAddress(_HWPriv),
        kRegisterBased
            | RESULT_SIZE (SIZE_CODE (sizeof (OSErr)))
            | REGISTER_RESULT_LOCATION (kRegisterD0)
            | REGISTER_ROUTINE_PARAMETER(1,kRegisterA0,
                SIZE_CODE(sizeof(address)))
            | REGISTER_ROUTINE_PARAMETER(2,kRegisterA1,
                SIZE_CODE(sizeof(count)))
        address,count);    // parameter(s)
}
```

The following API is missing from both `InterfaceLib` & the headers! (Bonus for reading this far!)

```
pascal ComponentResult TVSetFrequency(TVTunerComponent ci, long frequency)
//FIVEWORDINLINE(0x2F3C, 0x04, kSelectTVSetFrequency, 0x7000, 0xA82A);
//        MOVE.L    #$00040001,-(A7)
//        MOVEQ    #$00,D0
//        _ComponentDispatch
{
    return (ComponentResult) CallUniversalProc(
        (UniversalProcPtr) GetToolboxTrapAddress(0xA82A),
        kD0DispatchedPascalStackBased |
            RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult))) |
            DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode) |
            DISPATCHED_STACK_ROUTINE_PARAMETER( 1,
                SIZE_CODE(sizeof(TVTunerComponent))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER( 2,
                SIZE_CODE(sizeof(long))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER( 3,
                SIZE_CODE(sizeof(long))),
        0x0000,        // D0 selector
        ci, frequency,    // parameter(s)
        0x00040001);    // Stack dispatched selector
}
```

Back to top

Full Example: HFS+

Here I present a complete glue file for the eXtended `DiskInit` routines for HFS+:

```
/*
    File:        DiskInit.Glue.c

    Copyright:    © 1984-1997 by Apple Computer, Inc.
                All rights reserved.
*/

#include <DiskInit.h>
#include <MixedMode.h>

static UniversalProcPtr gPack2TrapUPP = kUnresolvedCFragSymbolAddress;
static UniversalProcPtr gUnimplementedUPP = kUnresolvedCFragSymbolAddress;

pascal OSErr DIXFormat(short drvNum, Boolean fmtFlag,
                unsigned long fmtArg, unsigned long *actSize)
// THREEWORDINLINE(0x700C, 0x3F00, 0xA9E9);
//        MOVEQ    #$0C,D0    | 700C
//        MOVE.W    D0,-(A7)    | 3F00
//        _Pack2            | A9E9
{
    long    private_result = unimpErr;    // assume unimplemented A-Trap

    if ((Ptr) gUnimplementedUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gUnimplementedUPP = GetToolboxTrapAddress(_Unimplemented);

    if ((Ptr) gPack2TrapUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gPack2TrapUPP = GetToolboxTrapAddress(0xA9E9);

    if ((Ptr) gPack2TrapUPP != (Ptr) gUnimplementedUPP)
    {
    private_result = CallUniversalProc(gPack2TrapUPP,
        kStackDispatchedPascalStackBased |
            RESULT_SIZE(SIZE_CODE(sizeof(OSErr))) |
            DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(1,
                SIZE_CODE(sizeof(short))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(2,
                SIZE_CODE(sizeof(Boolean))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(3,
                SIZE_CODE(sizeof(unsigned long))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(4,
                SIZE_CODE(sizeof(unsigned long))),
        0x000C,                // selector
        drvNum, fmtFlag, fmtArg, actSize);    // parameter(s)
    }
    return (OSErr) private_result;
}

pascal OSErr DIXZero(short drvNum, ConstStr255Param volName,
    short fsid, short mediaStatus, short volTypeSelector,
    unsigned long volSize, void *extendedInfoPtr)
// THREEWORDINLINE(0x700E, 0x3F00, 0xA9E9);
//        MOVEQ    #$0E,D0        | 700E
//        MOVE.W    D0,-(A7)    | 3F00
```

```
//        _Pack2                | A9E9
{
    long    private_result = unimpErr;    // assume unimplemented A-Trap

    if ((Ptr) gUnimplementedUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gUnimplementedUPP = GetToolboxTrapAddress(_Unimplemented);

    if ((Ptr) gPack2TrapUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gPack2TrapUPP = GetToolboxTrapAddress(0xA9E9);

    if ((Ptr) gPack2TrapUPP != (Ptr) gUnimplementedUPP)
    {
    private_result = CallUniversalProc(gPack2TrapUPP,
        kStackDispatchedPascalStackBased |
            RESULT_SIZE(SIZE_CODE(sizeof(OSErr))) |
            DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(1,
                SIZE_CODE(sizeof(short))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(2,
                SIZE_CODE(sizeof(ConstStr255Param))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(3,
                SIZE_CODE(sizeof(short))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(4,
                SIZE_CODE(sizeof(short))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(5,
                SIZE_CODE(sizeof(short))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(6,
                SIZE_CODE(sizeof(unsigned long))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(7,
                SIZE_CODE(sizeof(void *))),
        0x000E,                         // selector
        drvNum, volName, fsid, mediaStatus,        // parameter(s)
        volTypeSelector, volSize, extendedInfoPtr);
    }
    return (OSErr) private_result;
}

pascal OSErr DIReformat(short drvNum, short fsid,
                ConstStr255Param volName, ConstStr255Param msgText)
// THREEWORDINLINE(0x7010, 0x3F00, 0xA9E9);
//        MOVEQ    #$10,D0        | 7010
//        MOVE.W    D0,-(A7)    | 3F00
//        _Pack2                | A9E9
{
    long    private_result = unimpErr;    // assume unimplemented A-Trap

    if ((Ptr) gUnimplementedUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gUnimplementedUPP = GetToolboxTrapAddress(_Unimplemented);

    if ((Ptr) gPack2TrapUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gPack2TrapUPP = GetToolboxTrapAddress(0xA9E9);

    if ((Ptr) gPack2TrapUPP != (Ptr) gUnimplementedUPP)
    {
```

```
    private_result = CallUniversalProc(gPack2TrapUPP,
        kStackDispatchedPascalStackBased |
            RESULT_SIZE(SIZE_CODE(sizeof(OSErr))) |
            DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(1,
                SIZE_CODE(sizeof(short))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(2,
                SIZE_CODE(sizeof(short))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(3,
                SIZE_CODE(sizeof(ConstStr255Param))) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(4,
                SIZE_CODE(sizeof(ConstStr255Param))),
        0x0010,                 // selector
        drvNum, fsid, volName, msgText);    // parameter(s)
    }
    return (OSErr) private_result;
}
```

Back to top

Full Example: ControlStrip

```
/*
     File:        ControlStrip.Glue.c

     Copyright:    © 1984-1997 by Apple Computer, Inc.
                  All rights reserved.
*/

#include <MixedMode.h>
#include <ControlStrip.h>

static UniversalProcPtr gControlStripTrapUPP = kUnresolvedCFragSymbolAddress;
static UniversalProcPtr gUnimplementedUPP = kUnresolvedCFragSymbolAddress;

#define _ControlStripDispatch 0xAAF2

pascal Boolean SBIsControlStripVisible(void)
// TWOWORDINLINE(0x7000, 0xAAF2);
//    MOVEQ    #$00,D0
//    _ControlStripDispatch
{
    long    private_result = 0L;

    if ((Ptr) gUnimplementedUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gUnimplementedUPP =
            GetToolboxTrapAddress(_Unimplemented);

    if ((Ptr) gControlStripTrapUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gControlStripTrapUPP =
```

```
            GetToolboxTrapAddress(_ControlStripDispatch);

    if ((Ptr) gControlStripTrapUPP != (Ptr) gUnimplementedUPP)
    {
        private_result = CallUniversalProc(gControlStripTrapUPP,
            kD0DispatchedPascalStackBased |
                DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode) |
                RESULT_SIZE(SIZE_CODE(sizeof(Boolean))),
            0x0000);    // selector
    }
    return (Boolean) private_result;
}

pascal void SBShowHideControlStrip(Boolean showIt)
// THREEWORDINLINE(0x303C, 0x0101, 0xAAF2);
//    MOVE.W    #$0101,D0
//    _ControlStripDispatch
{
    if ((Ptr) gUnimplementedUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gUnimplementedUPP = GetToolboxTrapAddress(_Unimplemented);

    if ((Ptr) gControlStripTrapUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gControlStripTrapUPP = GetToolboxTrapAddress(_ControlStripDispatch);

    if ((Ptr) gControlStripTrapUPP != (Ptr) gUnimplementedUPP)
    {
        CallUniversalProc(gControlStripTrapUPP,
            kD0DispatchedPascalStackBased |
            DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode) |
            DISPATCHED_STACK_ROUTINE_PARAMETER(1,
                SIZE_CODE(sizeof(Boolean))),
            0x0101,    // selector
            showIt);    // parameter(s)
    }
}
```

Back to top

Full Example: Power Manager

```
/*
     File:        Power.Glue.c

     Copyright:    © 1984-1997 by Apple Computer, Inc.
                  All rights reserved.
*/

#include <MixedMode.h>
#include <Power.h>
```

```
static UniversalProcPtr gPowerTrapUPP = kUnresolvedCFragSymbolAddress;
static UniversalProcPtr gUnimplementedUPP = kUnresolvedCFragSymbolAddress;

pascal Boolean HardDiskPowered(void)
// TWOWORDINLINE(0x7006, 0xA09E);
//    MOVEQ    #$06,D0        | 7006    ; Move selector
//    _PowerMgr            | A09E    ; for _HardDiskPowered
{
    long    private_result = 0L;

    if ((Ptr) gUnimplementedUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gUnimplementedUPP = GetToolboxTrapAddress(_Unimplemented);

    if ((Ptr) gPowerTrapUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gPowerTrapUPP = GetToolboxTrapAddress(0xA09E);

    if ((Ptr) gPowerTrapUPP != (Ptr) gUnimplementedUPP)
    {
        private_result = CallUniversalProc(gPowerTrapUPP,
            kD0DispatchedPascalStackBased |
                DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode) |
                RESULT_SIZE(SIZE_CODE(sizeof(Boolean))),
            0x0006);    // selector
    }
    return (OSErr) private_result;
}

pascal void SpinDownHardDisk(void)
// TWOWORDINLINE(0x7007, 0xA09E);
//    MOVEQ    #$07,D0        | 7006    ; Move selector
//    _PowerMgr            | A09E    ; for _SpinDownHardDisk
{
    if ((Ptr) gUnimplementedUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gUnimplementedUPP = GetToolboxTrapAddress(_Unimplemented);

    if ((Ptr) gPowerTrapUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gPowerTrapUPP = GetToolboxTrapAddress(0xA09E);

    if ((Ptr) gPowerTrapUPP != (Ptr) gUnimplementedUPP)
    {
        CallUniversalProc(gPowerTrapUPP,
            kD0DispatchedPascalStackBased |
                DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode),
            0x0007);    // selector
    }
}

pascal Boolean IsSpindownDisabled(void)
// TWOWORDINLINE(0x7008, 0xA09E);
//    MOVEQ    #$08,D0        ; Move selector
//    _PowerMgr            ; for _IsSpindownDisabled
{
    long    private_result = 0L;
```

```
    if ((Ptr) gUnimplementedUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gUnimplementedUPP = GetToolboxTrapAddress(_Unimplemented);

    if ((Ptr) gPowerTrapUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gPowerTrapUPP = GetToolboxTrapAddress(0xA09E);

    if ((Ptr) gPowerTrapUPP != (Ptr) gUnimplementedUPP)
    {
        private_result = CallUniversalProc(gPowerTrapUPP,
            kD0DispatchedPascalStackBased |
            DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode) |
            RESULT_SIZE(SIZE_CODE(sizeof(Boolean))),
            0x0008);    // selector
    }
    return (OSErr) private_result;
}

// These two API's are strange in that they pass the boolean to the
//A-Trap via the high-word of D0. The low-word holds the selector.
//(Is this weird or what?)

pascal void SetSpindownDisable(Boolean setDisable)
// FOURWORDINLINE(0x4840, 0x303C, 0x0009, 0xA09E);
//    SWAP    D0        ; Move setDisable to high-Word of D0
//    MOVE.W    #$0009,D0    ; Move selector to low-Word of D0
//    _PowerMgr        ; for _SetSpindownDisable
{
    if ((Ptr) gUnimplementedUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gUnimplementedUPP = GetToolboxTrapAddress(_Unimplemented);

    if ((Ptr) gPowerTrapUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gPowerTrapUPP = GetToolboxTrapAddress(0xA09E);

    if ((Ptr) gPowerTrapUPP != (Ptr) gUnimplementedUPP)
        CallUniversalProc(gPowerTrapUPP,
            kD0DispatchedPascalStackBased |
                DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode),
            (setDisable << 16) | 0x0009);    // selector
}

pascal void AutoSleepControl(Boolean enableSleep)
// FOURWORDINLINE(0x4840, 0x303C, 0x000D, 0xA09E);
//    SWAP    D0        ; Move setDisable to high-Word of D0
//    MOVE.W    #$000D,D0    ; Move selector to low-Word of D0
//    _PowerMgr        ; for _AutoSleepControl
{
    if ((Ptr) gUnimplementedUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gUnimplementedUPP = GetToolboxTrapAddress(_Unimplemented);

    if ((Ptr) gPowerTrapUPP == (Ptr) kUnresolvedCFragSymbolAddress)
        gPowerTrapUPP = GetToolboxTrapAddress(0xA09E);

    if ((Ptr) gPowerTrapUPP != (Ptr) gUnimplementedUPP)
```

```
        CallUniversalProc(gPowerTrapUPP,
            kD0DispatchedPascalStackBased |
                DISPATCHED_STACK_ROUTINE_SELECTOR_SIZE(kTwoByteCode),
            (enableSleep << 16) | 0x000D);    // selector
}
```

Back to top

Summary

Now that you know how to glue in all those routines missing from `InterfaceLib`, you are now officially out of excuses. Time to write the next killer CFM application!

Back to top

References

Inside Macintosh: PowerPC System Software, Chapter 2 Mixed Mode Manager

"A Fragment of Your Imagination", Joe Zobkiw, ISBN:0-201-48358.

Back to top

Downloadables

Acrobat version of this Note (K). Download

Back to top
