Updating Open-PowerPlant — Why Now?

Rick Aurbach
Aurbach & Associates, Inc.

August 29, 2011

Abstract

The September 5, 2011 update to the Open-PowerPlant project has as its main goal, the release of Constructor as a Universal Binary application. As a side effect, a number of changes to the PowerPlant framework were also made. This note documents the changes and provides a rationale for the work.

Contents

Introduction

Ok, I admit it. I really enjoy programming in Carbon / C++ / PowerPlant. I like it so much that my Windows products are written using a custom framework that implements the PowerPlant architecture and enables me to write code that works on both platforms and which provide customers with equal functionality, regardless of their platform of choice.

In fact, PowerPlant and this ability to write and maintain a single code-base for both platforms has been a critical in allowing my one-programmer 'shop' to develop and support multiple products, including multi-application program suites.

But the writing has been on the wall for a long time. Apple has chosen to move in other directions. And we all know that Carbon is dead and that as a result PowerPlant has no future. So why am I updating Open-PowerPlant now? Simply because

- I need to support my existing products while I am rewriting them from scratch in Cocoa, and
- Since Constructor is an old PowerPC application, it won't run on Lion.[1]

So this update to Open-PowerPlant includes a new, Universal Binary release of Constructor (both sources and as a ready-to-run application bundle).

I particularly want to thank my beta-tester, Bryan Christianson, who identified a number of critical errors in my work (particularly under Lion). Without him, this work would not have been nearly as successful.

But there's more... in the course of this work, I also made some enhancements to PowerPlant itself.

In the sections which follow, I'll discuss the PowerPlant changes. But first, I'd like to share with you the premises which guided this work.

My Upgrade Premises

The work I'm describing here is based on the following set of premises:

- ▶ The Classic environment is dead and needs not be resurrected. It is safe to treat PowerPlant as a MachO / Carbon / Mac OSX -only framework. But there is no benefit to be gained by going through the PowerPlant code and systematically removing Classic-related functionality.
- ▶ Even though it has some nice features I hate to give up, there is no good reason to avoid the transition from the CodeWarrior IDE to Xcode. My development environment for PowerPlant applications is Xcode 3.2.x.
- ▶ Since Xcode 3.2.x does not support pre-10.4 environments, neither should PowerPlant. It's okay to use APIs introduced in 10.4. But when we do, it is also kind to leave the 10.3-compatible code behind (commented out). After all, there may be some folks out there who need to support 10.3 (but don't ask me why).

[1] Actually, I can't find *any* resource editor that runs as an Universal Binary. I'm just starting to update `Rezilla`, so keep an eye out for it.

- ▶ Since PowerPlant has a very limited future, major architectural changes make no sense and should be avoided. My goal is to update PowerPlant only so far as necessary to work nicely with current versions of Mac OSX.

  In particular, a partial list of what I've chosen *not* to do includes:

  - Rip out the Appearance bridge and the grayscale appearance classes.
  - Replace `FSSpec` throughout the framework with a more modern file representation (probably `CFURLRef`).
  - Replace Quickdraw throughout the framework with Quartz.
  - Rewrite the thread classes to use P-threads.
  - Replace the preference mechanism (i.e., `LPreferenceFile` and the resource classes) with `CFPreference`. I've done this in the Constructor code, but haven't migrated the changes back to the PowerPlant core.
  - Eliminate a number of other PowerPlant quirks and unnecessary complexity related to supporting the Classic environment.
  - Mine the Constructor code for more gems (beyond the code already in the Constructor Additions folder) and migrate them back into the PowerPlant core.

All nice to do, but it ain't going to happen.

Constructor Changes

The previous version of Constructor (v2.5.8) was an unbundled PowerPC application. The version of Constructor included here (v2.6) is a bundled, Universal Binary application.

That's the big change. In addition, there are a few minor improvements. The one you'll notice most directly is that the Text Traits editor now (once again) displays and edits style parameters.

One thing I deliberately did not change is where Constructor looks for custom types. That is, Constructor still looks for a Custom Types folder in the same folder as its application bundle. I did *not* move the Custom Types folder into the bundle.

PowerPlant Changes

The following sections summarize the main changes to PowerPlant in this release.

System Headers

Since we're focusing on Xcode rather than CodeWarrior, we've changes how system headers are accessed. In particular, we assume MachO-style headers, and have conditionalized all references to system headers.

Listing 1: Specifying System Headers

```
#ifndef __MACH__
    #include <system-header.h>
#endif
```

LStream Changes

The `LStream` class (and the related classes `LDataStream`, `LFileStream`, and `LHandleStream`) explicitly assume that the stream is big-endian and must be transparently converted to/from native-endian representations. This is usually a good idea, but not always, even when used to read and write resources.

By convention, resources are stored big-endian. This is not a problem for classic and PowerPC environments, since both of these processor families are also big-endian. But on Intel processors (which are little-endian), there is a problem.

Mac OSX globally installs a number of resource flippers — i.e., callbacks which perform endian conversions automatically. If a resource has an installed resource flipper (on either a global or application-wide basis), reading it on an Intel processor using the traditional `LStream` functions would cause an *additional*, unwanted endian swap, leading to errors.

Don't get carried away by this issue; Apple doesn't install all that many resource flippers. So mostly you don't have to worry about all this. But sometimes...

An example of this is problem is the `Txtr` (Text Traits) resource. There is a global resource flipper for this resource type. As a result, when read into memory it is automatically endian-converted on Intel systems.

We can verify this by looking at `UTextTraits::LoadTextTraits`

Listing 2: `UTextTraits::LoadTextTraits`

```
void
UTextTraits::LoadTextTraits(
    ResIDT              inTextTraitsID,
    TextTraitsRecord&   outTextTraits)
{
    TextTraitsH traitsH = LoadTextTraits(inTextTraitsID);
    // which just calls ::GetResource() -- no endian-aware code

    if (traitsH != nil) {
        ::BlockMoveData(*traitsH, &outTextTraits,
                        ::GetHandleSize((Handle) traitsH));
    } else {
        LoadSystemTraits(outTextTraits);
    }
}
```

which would never work on Intel without endian-conversion behind the scenes.

Now supposed that instead of this approach, you attempted to read the text traits data using stream operators.

Listing 3: Error: Reading TextTraits Data with a Stream

```
    StResource       rsrc{'Txtr', 123);
    LHandleStream    stream(rsrc);
    stream >> size;      // Wrong! Data flipped twice
    stream >> style;     // Wrong! Data flipped twice
        . . .
```

On Intel, the system will flip the data once when reading it into the StResource handle, and LStream will flip it again when extracting the data from the handle. The data will be flipped twice.

To prevent this from happening, I have added a boolean flag to LStream. If mNativeEndian is true, the data is assumed have the correct endian character and endian swaps will be omitted in the >> and << operators. By default, this flag is set to false, so the default operation of LStream is unchanged.

We also define the obvious accessor functions:

Listing 4: New LStream Accessors

```
    bool     IsNativeEndian () const
                 {    return mNativeEndian;        }

    void     SetNativeEndian (
                 bool          inNativeEndian = true )
                 {    mNativeEndian = inNativeEndian; }
```

Of course, to use this new feature effectively, we need a mechanism to determine whether a particular resource type has an installed resource flipper. Read on.

The UResFlip Namespace

I have modified UResourceMgr to add a new namespace, UResFlip. This namespace wraps the resource-flipper APIs.

Listing 5: The UResFlip namespace

```
namespace   UResFlip    {
    bool        HasFlipper (
                    ResType                 inResType );

    OSStatus    GetFlipper (
                    ResType                 inResType,
                    CoreEndianFlipProc *    outFlipProc,
                    void **                 outRefCon );

    OSStatus    SetFlipper (
                    ResType                 inResType,
                    CoreEndianFlipProc      inFlipProc,
                    void *                  inRefCon );

    OSStatus    Flip (
                    ResType                     inResType,
                    ResIDT                      inID,
                    void *                      ioData,
                    ByteCount                   inDataLen,
                    Boolean                     inIsNative  );
}
```

Unless you intend to implement your own resource flippers, the most important member of this namespace is HasFlipper. Using it, the previous code fragment can be written

Listing 6: Reading TextTraits Correctly

```
    StResource       rsrc{'Txtr', 123);
    LHandleStream    stream(rsrc);
    stream.SetNativeEndian(UResFlip::HasFlipper('Txtr'));

    stream >> size;     // This now returns valid data
    stream >> style;
        . . .
```

Standard Dialog Changes

In Standard Dialogs, PowerPlant implemented the `UNavServicesDialogs` module using a set of APIs which were deprecated and replaced in Mac OSX 10.4. These older APIs have some problems:

- Since the olds APIs use SFSpecs exclusively, they only support 31-character file names and the MacRoman character set. This is a general failing of PowerPlant as a whole, but is particularly problematic here.
- The old APIs are really oriented around using file type codes to associate file ownership with applications. In particular, the fact that they cannot deal with UTIs can be a problem in Lion when working with files which have only a file extension and no file type code. (They can fail to recognize that files without file type codes can be candidates for being opened.)

By changing `UNavServicesDialogs` to use more modern Navigation Services APIs, we can avoid these problems, with only a very minor API change.

Please note that these changes involve using APIs introduced in 10.4, so they will not work in pre-10.4 code. However, I've left the old code (commented out) in place so it is easy to restore the old behavior.

Trapping Classic Behavior

Since we no longer support the Classic environment and since there are minor API differences between Classic and the (new) Carbon APIs, I have made changes in UStandardDialogs.h, UClassicDialogs and UConditionalDialogs to make sure that UNavServicesDialogs is the only functioning flavor of the standard dialogs.

Changes to UNavServicesDialogs Namespace Functions

The UNavServicesDialogs namespace contains a number of simple functions as well as a number of classes.

The simple functions have been reimplemented without API changes:

Load This function used to call NavLoad, which is not implemented in Mac OSX. The function now does nothing, so it may be safely included (for the sake of compatibility with old code) in OSX-targeted applications.

Unload This function used to call NavUnload, which is not implemented in Mac OSX. The function now does nothing, so it may be safely included (for the sake of compatibility with old code) in OSX-targeted applications.

AskSaveChanges This function has been reimplemented using NavCreateAskSaveChangesDialog without an external API change.

AskConfirmRevert This function has been reimplemented using NavCreateAskDiscardChangesDialog wintout an external API change.

AskOpenOneFile This function uses LFileChooser internally and has been modified accordingly.

AskChooseOneFile This function uses LFileChooser internally and has been modified accordingly.

AskChooseFolder This function uses LFileChooser internally and has been modified accordingly.

AskChooseVolume This function uses LFileChooser internally and has been modified accordingly.

AskSaveFile This function uses LFileDesignator internally and has been modified accordingly.

Changes to LFileChooser

Substantial portions of LFileChooser were rewritten to use the Navigation Services APIs introduced in Mac OSX 10.4 and to provide support for UTIs. The following changes are important:

GetDialogOptions Previously, this function returned a pointer to a NavDialogOptions structure. Now it returns a pointer to a NavDialogCreationOptions structure.

In particular, note that the flags field of the NavDialogCreationOptions structure is named **optionFlags**, not dialogOptionFlags.

GetFileSpec Previously, there was only one version of this method, which returns an FSSpec&. Now, in addition to this method, there are two additional overload methods, returning FSRef* and CFURLRef* respectively.

UTI Support New APIs have been added to support UTIs. The LFileChooser class now maintains a CFArray of UTIs as an internal instance variable. This list can either be constructed automatically (based on the specified file type list) or explicitly.

GetUTIList Return the current UTI list. This routine uses "get semantics" — that is, it returns the actual UTI list data member and remains owned by LFileChooser.

CopyUTIList Replace the internal UTI list with the specified list.

AddUTIToList Add a single UTI to the internal UTI list.

AddUTIListToList Add an array of UTIs to the internal UTI list.

DeriveUTIList Create an internal UTI list by adding an array of UTIs for each of the file types specified by an LFileTypeList.

Listing 7: Summary of LFileChooser API changes

```
NavDialogCreationOptions *  GetDialogOptions();

void            GetFileSpec(
                    SInt32              inIndex,
                    FSSpec&             outFileSpec) const;

void            GetFileSpec (
                    SInt32              inIndex,
                    FSRef *             outFSRef )  const;

void            GetFileSpec (
                    SInt32              inIndex,
                    CFURLRef *          outURLRef ) const;

CFArrayRef      GetUTIList () const;

void            CopyUTIList (
                CFArrayRef              inList  );

void            AddUTIToList (
                CFStringRef             inUTI  );

void            AddUTIListToList (
                CFArrayRef              inUTIList  );

void            DeriveUTIList (
                const LFileTypeList&    inFileTypes  );
```

Please note that if you do not explicity set up the internal UTI list (using the new APIs) prior to invoking an OpenFile or ChooseFile dialog, the internal UTI list will be empty. In this case, the code will *automatically* call DeriveUTIList for you.

This means that in most circumstances, you need make no code changes (other than setting up your UTIs in your `info.plist` file) to get UTI processing.

Changes to LFileDesignator

Substantial portions of LFileDesignator were rewritten to use the Navigation Services APIs introduced in Mac OSX 10.4. The following changes are important:

GetDialogOptions Previously, this function returned a pointer to a NavDialogOptions structure. Now it returns a pointer to a NavDialogCreationOptions structure.

In particular, note that the flags field of the NavDialogCreationOptions structure is named **optionFlags**, not dialogOptionFlags.

GetFileSpec Previously, there was only one version of this method, which returns an FSSpec&. Now, in addition to this method, there are two additional overload methods. One returns an FSRef* and a CFStringRef*, the other returns a CFURLRef*.

Listing 8: Summary of LFileDesignator API changes

```
NavDialogCreationOptions *  GetDialogOptions();

void        GetFileSpec(
                FSSpec&              outFileSpec) const;

void        GetFileSpec (
                FSRef *              outParentRef,
                CFStringRef *        outFileName )    const;

void        GetFileSpec (
                CFURLRef *           outURLRef ) const;
```

Changes to StNavReplyRecord

They way NavReplyRecords are used has changed. Previously, the record was created and passed into the Navigation Services dialog function (NavGetFile, NavPutFile, etc.). The new dialogs don't do that. Instead, users extract a new NavReplyRecord from the Nav Services dialog when it completes successfully.

To accommodate this change in lifetime processing of NavReplyRecords, we needed to update the existing record (contained in the StNavReplyRecord class) when appropriate and be more careful with how and when that record was disposed. I made two changes:

- I added an mAllocated flag which is set to mean that a NavReplyRecord has been allocated via a call to NavDialogGetReply and needs to be explicitly disposed. This flag is used, for example, in the class destructor to determine whether or not to call NavDisposeReply.
- I added the UpdateReply method which calls NavDialogGetReply and sets mAllocated.

Using the new Navigation Services APIs introduced in Mac OSX 10.4, the Put File dialog no longer returns a descriptor to the new file specification in the reply record's selection field. Instead, it stores a descriptor to the new file's parent directory there, and stores the new file's name in the saveFileName field.

To account for these changes, I have overloaded the GetFileSpec method and created a CopySaveFileName method.

Listing 9: New StNavReplyRecord Output Methods

```
void            GetFileSpec(
                    FSSpec&      outFileSpec) const;

void            GetFileSpec(
                    FSRef*       outFileSpec) const;

CFStringRef     CopySaveFileName () const;
```

The latter method returns a copy of the internal data which is owned (and must be released) by the caller.

UQuickTime

The UQuickTime::GetMovieFromFile function uses LFileChooser and was updated to incorporate the above changes.