An Experience in Carbonization

Abstract:

This paper talks about a hand on experience of carbonization of an application, discussing in details the actual steps involved and the issues faced during the process. For general understanding of Carbonization refer to "Mac OS X, Your Existing Applications and Carbon" (Mindfire Solutions).

INTRODUCTION

With the release of Mac OS X, everybody wants to make to sure that his application runs on the new OS, and runs natively not in the Classic Environment.

OSX support three development approaches, Cocoa, Java & Carbon. Cocoa being the native Mac OSX development framework, may be the first choice for from scratch development (may be, because cocoa application will only run on OS X, which means different versions will have to be maintained for OS X and OS9)

The other development option is Carbon, which is most suited for porting of an existing application due to the following facts:

?? The Carbon APIs are based on existing Mac OS APIs including most of the (about 95%) existing Mac OS APIs, which a typical application uses. Hence, converting to Carbon is a straightforward process

?? The Carbon applications can run on Mac OSX as well as previous versions of the Mac OS (8.1 or later), which means single code base to maintain.

Although Carbon APIs are based on existing Mac OS APIs, there still are many issues to be considered during the porting process, mainly due to the architectural changes in OSX from previous Mac OS versions. The *Carbon Porting Guide.pdf,* available at the Apple's site or with the Carbon SDK, is the detailed document giving step-by-step process and discussing the issues in converting an existing application to carbon application.

While the above document talks about the steps involved and general issues faced in carbonization of a typical application running on earlier versions of Mac OS 9, what I am going to present here is the actual steps involved and the important issues faced in carbonization of a particular application, which was completed a few weeks ago.

FACT SHEET OF THE EXISTING APPLICATION

Type of application: Desktop Application
Size: About 100,000 loc
Project: Code Warrior Pro5
IDE: Code Warrior IDE4.0
FrameWork/Libraries: PowerPlant 2.0, WASTE Text Engine1.3
Resource Editor: PP Constructor 2.4.5
Interface Version: Universal Headers 3.2

The application in discussion was simulation software, having both complex processing and a highly graphical interface.

The main subsystems of the application were
- ?? A graphical Interface for showing simulation and with different types of Graphs associated with it

- ?? An UI editor with option for Drag n drop, moving/resizing of different components to make new models

- ?? File I/O for saving and loading of the models

- ?? Web and Email Integration

- ?? Online Tests integrated with models

- ?? Comprehensive Online library with Content/Index/Search options

• *Porting Environment*

Development Environment: Mac OS 9.0.3
CarbonLib SDK: Version 1.3.1
Tested on: Mac OS 8.6, 9.0.3, 9.1 and Mac OS X

Duration: 3 weeks

IN PREPARATION

- ### *No 68K Dependencies*
Mac OS X requires 100% native PPC code, so we need to remove any dependencies on 68K instruction. Fortunately, our application didn't have any of those and hence no steps here.

- ### *Carbon Dater*
To start with, Apple has provided a tool called Carbon Dater to analyze ones existing application/libraries for Carbon Compatibility. Carbon Dater produces .CCT file which needed to be mailed to carbondating@apple.com where after comparing it with the API database Apple has made, an html format report is mailed back, which can be used to have information about the scope of the efforts involved in the conversion.

Result of running Carbon Dater:

Supported APIs	-	85.3%
Supported with Modifications	-	0.7%
Supported But Not Recommended	-	4.8%
Unsupported API	-	9.2%

That meant 14.7 % of the code needed modifications, but the effective modification effort was reduced by the fact that these changes involved two different things. Getting the carbonized versions of the 3rd Party libraries (PP & WASTE here) and then making necessary changes to our own code.

The latest carbonized versions available of the above two libraries were **PowerPlant 2.1** and **WASTE 2.0**. PP2.1 was not available separately rather was being shipped with new release of Metrowerks **CodeWarrior 6**.

- ### *Updating to Current Universal Interface*
Although it isn't a requirement, doing so makes the transition easier. For us, it was the same step as moving our application to CW6, since it includes Universal Interface 3.3.2.

- ### *Moving the Existing Project to CW6*
This effectively meant multiple transitions.
> *CW IDE 4.0 to 4.1*
> *PowerPlant Constructor 2.4.5 to 2.5*
> *PP 2.0 to PP 2.1*
> *Universal Interface 3.2 to 3.3.2*

The first two were just matter of installing CW6 and then clicking the earlier project file. The CodeWarrior's Conversion Wizard automatically converted the project to the current version after confirmation.

By that time Apple was out with **Universal Interface 3.4**, so it was natural to update the Universal Interface 3.3.2 included with CW6. This had some implications, though, on building PowerPlant source code especially with *ACCESSORS_ARE_FUNCTIONS* flag set to TRUE (see *Opaque Data Structure* below). This was solved by making some modifications here and there in PP Code, mainly related to casting of *GrafPtr* & *CGrafPtr* and use of old/new API names.

• *Updating to Latest WASTE version (2.0)*
   Waste 1.3 to Waste 2.0
   CWASTEEdit to WTextView & Wtext

Although, the transition from WASTE 1.3 to WASTE 2.0b3 (which is carbonated) meant just replacing the libraries, the real change was concerning the wrapper class used (*CWASTEEdit*). *CWASTEEdit* was split into two classes; *WText*, which wraps all the public WASTE calls and maintains the *WEReference* handle and *WTextView*, a PPlant view, which inherits from *WText* and implements the interface between WASTE and PPlant. Plus the Ppob resource for the *WTextView*, has also been changed (Class ID NWSt). This meant replacing *CWASTEEdit* with *WTextView*, wherever it had been used (which was quite a lot) in the code and changing the Class ID in all the 'ppob' resources using *CWASTEEdit*.

The approach taken was, write a new *CWASTEEdit* class derived from new *WTextView* class, give it the same Class ID as the old *CWASTEEdit* class. Then add the appropriate interface for those functions which are missing from *WTextView* (like defining our own *CWASTEEdit::InsertPtr()*, which just calls the corresponding function *WTextView::InsertText()*). This enabled us to retain all the existing *CWASTEEdit* related code and resource.

With all the dependencies taken care of, the time was to attack our own code. The aim would be to try to maintain the same code base for Classic and Carbon Target and it was surprisingly easy.

• *Opaque Data Structures*
One of the major changes in Carbon is that it limits direct application access to some Mac OS data structures. These include *WindowPort*, *GrafPort*, *QDGlobals* and many others frequently used data structures. These data structures are called Opaque Data Structure and Carbon uses different Accessor Functions to get/set their values. These accessor functions are also available as a static library *CarbonAccessors.o*, an application linking against *InterfaceLib* and other non-Carbon libraries can continue to build with them and

still use the Accessor Functions defined in *CarbonAccessor.o*. This meant, a few more steps forward towards Carbonizing the code without actually separating the classic and carbon target.

The interfaces for these accessor functions are available through Universal Interface if the flag *ACCESSOR_CALLS_ARE_FUNCTIONS* is set to 1.

Some of the accessor functions are also defined in PPlant's *UTBAccessor.h* file inline, when the above flag is 0 (By the way, It results in 'illegal function overloading' error for three accessor functions already defined in *MacWindows.h* with Universal Interface 3.4)

As the next step *CarbonAccessor.o* was added to the existing classic link and each of the source code was modified to use Carbon accessor functions, with the following conditional macro at the top.
> *#define ACCESSOR_CALLS_ARE_FUNCTIONS 1*

Well, this wasn't the case actually, rather the above line was added directly in the existing precompiled header (.pch) which gave us the access to functions across all the source file, but with an implication. On adding the above line to the precompiled header, the content of PPlant's *UTBAccessors.h* was ignored and some of the PP file refused to compile. With a little change in the above file, everything settled.

The above macro gave access to accessor functions, but this didn't ensure that none of the Opaque toolbox data structures is being used any more. For this another macro is used
> *#define OPAQUE_TOOLBOX_STRUCTS 1*

This couldn't be placed in precompiled header file because PP still uses Opaque data structures directly for non-carbon targets.

At this point, none of the code used Opaque Toolbox data structures, but by calling the accessor functions in *CarbonAccesor.o* (or some from PP's *UTBAccessors.h*). The application still linked against the classic libraries and it will run on any Mac OS release as it used to do, because it doesn't require the *CarbonLib* at runtime.

CLASSIC AND CARBON TARGETS

Time to move separately for Classic and Carbon targets.

• *Preparing Carbon Target*

Preparing a Carbon target was just about creating a new target by copying the existing Classic target from the *Targets* tab of CW IDE. Precompiled headers were replaced with appropriate 'Precompiled headers for Carbon'. *InterfaceLib* and other classic libraries were removed from the new target giving way to *CarbonLib*. *CarbonAccessor.o* was also no longer needed in presence of *CarbonLib*.

For carbon targets, PP defines the following macros in its precompiled header file.

```
#define PP_Target_Carbon            1
#define PP_Target_Classic           (!PP_Target_Carbon)
#define TARGET_API_MAC_CARBON       PP_Target_Carbon
#define TARGET_API_MAC_OS8          PP_Target_Classic
```

TARGET_API_MAC_CARBON macro is used by the Universal Interface header files to decide about the API set available to the Carbon target.

There were some compilation problems with the PP source files for the above carbon target, mainly related to typecasting and use of some older routines and macro calls to the mixed mode manager, which had to be replaced with UPP Accessor Functions.

• *Renamed / Modified API's*

Again, these were surprisingly few because most of the code used PP's utility wrapper classes instead of direct calls. With PP carbonized, the task was made easier.

Two non-supported APIs, which were used mostly in the code were, well, *c2pstr* and *p2cstr*. One solution was to write them of our own calling the new *c2pstrcpy* and *p2cstrcpy* (they take two arguments, source and destination pointers) from them. Easier way though, was to include the following lines for carbon target in the some prefix file, which makes the older APIs available to the carbon target as macros (again no actual change in the code).

```
#if PP_Target_Carbon
    #define OLDP2C      1
#endif
```

Some of the older menu-handling APIs are not available in carbon so, they needed to be replaced with the corresponding APIs (*EnableItem* by *EnableMenuItem*). These Menu related codes were mostly at one place instead of being spread all along, hence it was easier just replacing them with the newer calls in-place. These changes were valid for classic target also because replacing APIs are available in *MenusLib 8.5* and later.

Some of the scrap related APIs have also changed in carbon, but again, replacing the direct API call with PP's *UScrap* scarp utility functions, did the task.

• *Standard File Dialogs vs Navigation Services*

In carbon Navigations services replaces the standard File Package. In PP terms, it meant removing *UConditionalDialogs.cp* and *UClassicDialogs.cp* from the carbon target and adding *UNavServicesDialogs.cp*. Code wise, all file handling using *UStandardFiles* and *UConditionalDialogs* had to be replaced conditionally (*#if PP_Target_Carbon*) with *UNavServicesDialogs*.

Apart from carbon compatibility, Navigation services offer some enhancements over Standard File Package, so using it for classic targets whenever possible, is a good thing.

• *Printing*

Carbon has a new Printing Manager defining set of APIs, which replaces that of original Printing Manager. Carbon Printing Manager allows applications to print both on Mac OS 8 & 9 with existing printer drivers and on Mac OS X with new printer drivers.

Our code was using *UPrintingMgr* class from PP to perform the printing task, which is included in the list of **will be obselete** files in PP 2.1. The new interface for printing in PP 2.1 is *UPrinting.h,* which has three different implementation files:
  UClassicPrinting, *UcarbonPrinting* and *USessionPrinting*

The new classes are *LPrintSpec* and *StPrintContext*, *StPrintSession* and *UPrinting*. Replacing the *UPrintingMgr* with *UPrinting* worked for both classic and carbon targets and ensured same code base.

• *CD Detection, InterfaceLib vs IOKit*

Absence of Device Managers in Carbon made this task little difficult. The existing application was making calls to Device Manager, which is not a part of Carbon, as it cannot run on Mac OS X. The replacement, I/O Kit, is a Mac OS X technology, which cannot run on Mac OS 8 and 9. The solution, hence, was to conditionally fork the code and make calls to either, the Device Manager or I/O Kit, depending upon the platform it was running on.

Forking the code like this, presented some build issues, like while building for carbon, the Universal Interface conditionalized out any non-carbon functions (device manager's calls). Attempting to call any of these functions generated compile error indicating missing prototypes. The solution was to declare the appropriate prototypes as required (from I/O Kit from OS X and from device manager for earlier OS) in our own code and then dynamically loading the shared library and getting the addresses of the needed symbols. Like, for OS 8&9, *GetSharedLibrary* and *FindSymbol* were used respectively to load the *InterfaceLib* and obtain the function pointers for *GetDrvQHdr* & *PBStatusSync* (on OS X, I/O Kit was used in the same way).

• *Using Casting Functions*

Values of type *DialogPtr*, *WindowPtr* & *GrafPtr* can no longer be directly casted, but instead "Casting Functions" like *GetWindowPort(WindowPr port)* window should be used to obtain the value of one from another. Direct casting wouldn't affect compilation, but would cause crash on OS X.

At this stage, all the functional subsystems of the application were ready with all the necessary 'Carbon Compatible Modifications' and the next step was to build the carbon version of the application linked against *CarbonLib*. This new application would run on pre OS X releases of Mac OS (8.6 and later), when *CarbonLib* system extension is installed.

RUNNING ON OS X

Time for running it on OS X

• *Adding a 'plst' 0 Resource*

On Mac OS X, carbon applications, which do not contain *'plst'* 0 resource won't be recognized as carbon application. To ensure this, the application has to include a *'plst'* 0 or *'carb'* 0 resource.

With these, the application could now run on OS X as a carbon application, i.e. not in classic environment. It would have got the new look and feel of the aqua interface, translucent windows/menus, glowing rounded aqua buttons etc.

OS X SPECIFIC ISSUES

Although the application was able to run on Mac OS 8&9 and OS X both, many OS X specific issues still needed to be resolved.

• *GUI Tweaking*

Visually, Mac OS X is simply stunning with its Aqua Human Interface. Color, depth, transparency and animation are used to their full. To ensure that the new application looked good (ok, great!) on OS X, it needed some tweaking in the resource.

Though the development was going on OS 9, editing and tweaking of the resource was done on OS X directly using PP Constructor 2.5. The reason was to avoid the long cycle of changing the resource, building the application on OS 9, copying and running it on OS X machine, just to see the effects of the changes.

- ### *Conditional File Quit Menu*

Carbon application running on OS X, automatically have a Quit menu under the application menu, provided by the Aqua interface. Hence, the Quit menu added to File Menu in the application became redundant on X. However, because for Mac OS 8 and 9 this menu was still needed, it had to be removed from the menu conditionally, when the application was running on X.

> *MyApp::MakeMenuBar() {*
> *Lapplication::MakeMenuBar();*
> *if(UenvironMent::GetOSVersion() >= 0x00001000)*
> *// remove the File/Quit menu*
> *}*

- ### *Double Buffered Windows*

In Mac OS X, all windows are buffered, that is, a window's content is written first to a buffer, which is then periodically transferred to the screen by the Window Manager. This was actually a plus point and resulted in smoother graphics but for some cases. Like, if during one event processing, a rectangle was painted and then erased within a loop to give a blinking effect, then the painted rectangle never appeared. All drawing calls resulted in updating the contents of the buffer instead of the screen and during next event processing, only the final state of the buffer is transferred to the screen.

The first solution applied was to explicitly call *WaitNextEvent* with *updateMask* during each iteration of the loop, but that was rather slow. Preferred way was to use

> *QDFlushPortBuffer(currentPort, dirtyRegion);*

to flush the drawing to the screen immediately, whenever *QDIsPotBuffered(port)* returned true (returns false on Mac OS 8/9).

Further, if anything is drawn directly into a window's pixel map (*SetCPixel()* in our case), QuickDraw cannot tell which parts of the pixel map are dirty, so they may not be updated to the screen in next refresh. To work around, *QDFlushPortBuffer* should be called explicitly with nonempty region parameter describing the modified pixel.

- ### *Theme Brush and Background Color*

The following code resulted in a black rectangle on OS 8 and 9 but with Theme background brush on OS X, it wasn't the case.

> *RGBBackColor(&rgbBlack);*
> *EraseRect(&rect);*

The fix was to use the following code instead of the above calls.

> *RGBForeColor(&rgbBlack);*
> *PaintRect(&rect)*

- **LBevelButton, Indistinguishable Normal/Pushed state**

Due to the translucency on Mac OS X, it was difficult to distinguish the pushed state of a *'Sticky Bevel Button with picture'*, from that of normal un-pushed state even with *'large bevel size'*. So, different images had to be used for pushed and normal state of the button. Since, PP constructor has no option for this at the design time, a new *LBevelButton* derived class was included for dynamically changing the picture for normal/pushed state.

- **LTableView Hiliting Problem**

This was an interesting problem in the sense that it took us some time to realize that it's a problem (which was initially discarded as new Aqua interface feature). For all the *LTableView* used in the application, the hiliting was not proper. Instead of a hiliting rectangle with the current hiliting color, there were just a few evenly spaced horizontal lines with the hiliting color

The reason was, since *LTableView* didn't have its own background (it's a view), hiliting was being applied on the underlying window's theme background rather. On OS X, the background theme brush has horizontal lines pattern with some white line in between. Hiliting was applied for white part of the background only giving few horizontal lines. The solution was to give *LTableView* its own background color (white) by overriding it. With white background color for the tables, the proper hiliting rectangle was back as it used to be on Mac OS 8 and 9.

CONCLUSIONS

Although these were the issues related to a particular application, they are typical enough to give a general insight about the differences in technologies used. This could also be helpful for starters, if they don't want to straightforward dig into 164 pages of "*Carbon Porting Guide.pdf*". Nevertheless, if not anything else, somebody out there might be lucky enough to find his or her problem discussed here. (Nothing bad in being so optimistic after putting some efforts in writing these pages.)

Mindfire Solutions is an offshore software services company in India. Mindfire possesses expertise in multiple platforms, and has built a strong track record of delivery. Mindfire passionately believes in the power of porting and its many advantages for software product companies.

We have developed specialized techniques to make porting efficient and smooth, and to solve the issues specific to porting. We offer core development and QA/testing services for your porting requirements, as well as complete life-cycle support for porting.

If you want to explore the potential of porting, please drop us an email at info@mindfiresolutions.com. We will be glad to help you.

To know more about Mindfire Solutions, please visit us on www.mindfiresolutions.com