Mac OS-Windows Porting

Mindfire Solutions
www.mindfiresolutions.com

July 15, 2001

Abstract:

This paper talks about hand on experiences of porting of applications, involving Mac and Windows, discussing in detail the issues faced and solutions/strategies adopted during the process.

Introduction

Given the diversity of the present computing world in terms of different CPUs and Operating systems, software product companies are genuinely concerned about their presence on multiple platform. Added to this the fact that they already have a matured/evolved product on one platform, porting it to newer platforms seems to be the most logical solution, considering time and cost, for quickly expanding their existing marker base.

Taking into account that Microsoft's Windows has the largest installation base in the industry and Apple's Macintosh is the other popular platform with a strong product base, a lot of action is taking place in porting of applications involving these two platforms.

Process of porting an existing application from one platform to another often presents some issues and challenges. The focus of discussion for this paper, **"Mac - Windows Porting"** is no exception. Basically, this paper is gathering of different problems and their solutions compiled from the experiences of some of the porting projects completed at ***Mindfire***. Along with, it also outlines the best practices and strategies, to be used and cautions against the pitfalls and *"would be death traps"*, while porting of an application from Mac to Windows.

Although there are plenty of development options available on both the platform, for this paper, our discussion will be restricted to *Microsoft VC++* (*MFC*) on Windows and *Metrowerks' CodeWarrior* (*PowerPlant*) on Mac (two most popular development tools used for corresponding platforms).

As for any s/w projects, porting process is also unique in itselt. It is difficult to say that every points mentioned below will be relevant to your porting process. Consequently, the topics listed here are in no particular order.

General Issues Faced In Mac OS-Windows Porting

- ***Subclassing vs. Class ID***

The way GUI objects are created from the resource is different in MFC and PP. If you created a dialog from the resource template having a button control, then with MFC no CButton object will be created for the button control. Rather when you want to refer to that button control, you can either get its handle directly by calling

::GetDlgItem(parentDialogHandle, itemID)

or make a call to

parentDialogPtr->GetDlgItem(itemID).

This function returns a **CWnd** pointer pointing to a **CButton** object representing the button control. But there was no CButton object created initially, while creating the dialog, so where this CButton object comes from? Actually, the *GetDlgItem* call (and many such Win32 API calls, which returns pointer to CWnd, like *GetFocus*(), *GetCapture*() etc.) creates an appropriate MFC class object temporarily depending upon the type of GUI object, and attaches the specific GUI object's handle to that. Temporarily, because this MFC class object will be deleted upon the next system idle time after detaching GUI objects handle from it. This means that these returned pointers cannot be stored for later use.

With PP, the situation is different. PP uses Class ID linkage to associate the GUI object to the C++ Class objects. All GUI classes (say **LButton**) has to have a unique class id by including something like following in their class definition:

enum { class_ID = FOUR_CHAR_CODE('butn') }

It should further be registered with PowerPlant using PP's **TRegiterar** template class. Finally, the button resource created using the PP constructor will have a Class ID field assigned to the Class ID of C++ class whose object this button control should be linked with.
During the application execution, when PP is creating the dialog, it looks at the class ID of the control and matches it with the IDs of the classes registered with TRegisterar. On finding a match, it actually creates the C++ object of type *LButton* attaching the button control to that. Now, later when you want to refer to this button control you make a call to *parentDialog->FindPaneByID*() which returns the pointer to the actual LButton object. Unlike on Windows, this pointer points to permanent C++ object and can be stored for later use.

In fact the typical method is to store the pointers of the child controls by calling FindPaneByID() early during the dialog initialization (*LPand::FinishCreateSelf()*) and then later use this as and when required.

This difference in behavior presents a big, if not difficult, problem while porting a PP application to MFC. The straight forward approach seems to just replace all the **childControlPtr**'s (acquired by calling FindPaneByID) with **GetDlgItem(childControlID)**, but the mere count of occurrences of this situation translates it into a lot of task. Even doing a *search and replace* has the problem of changing **this***ChildControlPtr* to GetDlgItem(**that***ChildControlID*) accidentally and then you are left with a hard to find bug.

The better solution is to use **Dynamic Subclassing** on MFC. Subclassing is method on Windows to dynamically attach a control created from a dialog template to the desired **CWnd** object. When a control is dynamically subclassed, windows messages will route through the **CWnd**'s message map and call message handlers in the **CWnd**'s class first.

Messages that are passed to the base class will be passed to the default message handler in the control.

This means that changing the following code

```
MyDialog::FinishCreateSelf() {
        ……….
        mButton1Ptr = (LButton*)FindPaneByID(button1ID);
        ……….
        ……
}
```

to

```
MyDialog::OnInitDialog() {
        ……….
        mButton1Ptr =  new CButton;
        mButton1Ptr->SubclassDlgItem(button1ID, this);
        ……….
        ……
}
```

makes sure that you can keep all other code using mButtonPtr in its original form.

- ***CWnd::SetTimer() vs. LPeriodical::SpendTime()***

The typical use of timers in PP is to derive your concerned class with **LPeriodical** and then override its *SpendTime*() function. Then if you have started listening using *LPeriodical::StartListening()* call, your SpendTime function will be called during every iteration of the event processing loop (loop containing *WaitNextEvent*). Whereas on Windows you create a timer with unique ID and desired frequency by calling *CWnd::SetTimer*(…). You also have to providing address to your callback function, which should be called at the desired interval.

The difference however, is that on PP SpendTime is always called during the event processing loop, and there is no way to specify the time interval or frequency. The usual method is to maintain the current Tick Count (a tick is approximately 1/60 of a second) and then decide within the SpendTime function that whether the desired interval has elapsed or not. The solutions is to examine the code in SpendTime for the desired time interval, and then replace the *LPeriodical::StartListening*() and *LPeriodical::StopListening*() calls with corresponding *CWnd::SetTimer*() and *CWnd::KillTimer*(..) calls specifying the desired time interval.

- ***Little Endian vs. Big Endian***

The fact that Mac OS is a Big-endian platform and Microsoft Windows a Little-endian platform present many other issues due to different byte ordering. The typical of those are faced while performing stream operations with multiple byte data type. For example, if a particular application on Mac does File load and save of its data, all is well if the saved data files are used on Mac only. But trying to load the same data file with the windows version of program will fail because of the fact that the windows application will try to interpret the data according to its byte order convention.

One of the ways is to include the byte-order information in the data file header itself so as the target platform can accordingly adjust the byte order before reading. Better still is to store the data in file in particular byte-order, always fixed, irrespective of the platform it being used on. Say, when sticking to the Big-endian file format, you don't have to do anything special on Mac, and on Windows, you always invert the byte order before writing to or reading from the file.

- ***Individual Menu vs. Common Menu***

Mac has a common menu bar at the top, which dynamically changes to reflect the currently focused window. On Windows, where menus are attached with their corresponding windows itself, the closest to Mac behavior is that of a MDI application all the child windows shareing the main Frame Window menu. However, in that case the parent frame takes up all the desktop space hiding anything behind them, unlike Mac where desktop is visible behind the floating windows. This a typical decisional factor involved in porting any application from Mac to Windows. The options available are either to break up the common application menu bar into different window's specific menu and then attach them to their corresponding windows or to redesign the application as an MDI application.

- ***Full Path vs. FSSpec***

A file on Mac is represented as **FSSpec** structure with three parameters namely Volume reference number, directory ID of parent directly, and the filename. On Windows, it is represented as a flat string in form of *"[dirve:] parentFolderPath \ filename"*. One of the ways to handle this situation is to defined your own FSSpec structure on Windows as

```
typdef struct {
        short       vRefNum;
        long        parID;
        char        name[_MAX_PATH];
} FSSpec;
```

and then using the name field for all the purpose ignoring the vRefNum and parID fields. This ensures that the interface using FSSpec in the original code remains same on target code for Windows.

- ***String Resources: Flat vs. String sets***

String resource on Windows is flat whereas on Mac they are organized in String Set – String Number order. So, while porting the string resources to windows, it is suggested that the new string IDs should be generated according to the original *String Set-String Number* combination. For example, for a String Set with ID 1000 containing the strings numbered from 1 to 10, giving the corresponding windows string resource IDs from 100001 to 100010 can enable you to provide your own substitute for Mac API *GetIndString(char outStr[256], short strSet, short strNum)* as following.

```
void GetIndString(char outStr[256], short strSet, short strNum) {
        int strID = strSet*100 + strNum;
        ::LoadString(resourceModule, strID, outStr, 255);
}
```

- ***Enter/Exit MenuLoop***

On Mac, when any of the popup menus is being displayed (by clicking on the menu bar or something alike), the system enters in modal state and devotes all the time in tracking the shown menu. So during that time your application receives no time slice to execute. This seemingly shortcoming had put us in an interesting situation in one of our graphical simulation software port. With original application on Mac, if the menu was clicked while the simulation running, the simulation stopped until the menu is dismissed. After porting the same application to Windows, if a user clicked on the menu while the simulation was running, he was faced with irresponsiveness of the application (simulation being expensive task in terms of CPU usage). As a result, we had to explicitly write a few line of codes to pause the simulation during the menu tracking, by capturing the WM_ENTERMENULOOP and WM_EXITMENULOOPmessages. (By the way, just for reference, we had to capture WM_ENTERSIZELOOP and WM_EXITSIZELOOP also in the above situation)

Different Data Structures

Most of the commonly used data structures on Mac have their counterpart available on Windows, either as it is or with some minor differences. The differences are generally in the size of the fields used. For example, for a **Point** structure on Mac you have **POINT** structure available on Windows, the difference being that **Point** have two fields **v** and **h** defined as **short** and **POINT** have them defined as **long x** and **long y.** These data types are typically used at so many places that a commonly applicable strategy is needed for all of them.

The solution lies in abstracting these data types by encapsulating them into **Wrappers**, or to write conversion utility functions like **PointToPOINT/ POINTToPoint** (or implement them as macros).

Graphics Handling: GDI vs. QuickTime

- ***HDC vs. GrafPtr***

Graphics handling on Mac and Windows are different. *Win32 Graphics API* usually takes a handle to device context as their first parameter and then the required operation is performed on that device context. On the contrary, on Mac drawing is handled by **QuickDraw,** which maintains a pointer to the current **GrafPort**. Subsequent drawing calls draws on the current port. Another difference is that for Mac, **GrafPtr** and **WindowPtr** are interchangeable, i.e. a call to *SetGWorld(myWindowPtr, null),* will route the subsequent drawing request to the window pointed by **myWindowPtr**. Whereas for Windows, we have to explicitly acquire the device context handle from the window by a call to ::*GetDC(myWindowHandle),* and then do drawing operation on the windows by passing the acquired device handle as the first parameter of the drawing APIs.

- ***CDC vs. LDC??***

MFC has **CDC** class (and its derived classes), which encapsulate the device context handle giving a C++ interface for drawing operations. Such a class is missing from PowerPlant. This difference usually results in a lot of code changes on Mac and Windows.

A good strategy to handle both the above points is to write your own *wrapper* class for graphics handling. On Windows, this class can be exactly similar to CDC while on Mac it should maintain a pointer to its own GrafPtr. All the drawing operation performed using this class should reflect on the GrafPtr maintained by the class.

For an existing Mac code that uses direct QuickDraw calls, replacing them in place with corresponding GDI calls (using HDC or CDC), might be easier, but for cross developing application on Mac and Windows simultaneously, spending a little time in writing the graphics wrapper class can save a lot of time later. Also for porting projects, where maintaining single code base on Mac and Windows is required, the above-mentioned wrapper class is the most viable solution, even though it might mean initially changing the original Mac code to use your wrapper class in place of direct QD calls.

- ***COLORREF vs. RGBColor***

The color value of windows is represented in form of **COLORREF** data structure, which has long data type. The lower three bytes of this data structure respectively represents the red, green and blue component of the color, restricting the value of individual component from 0-255. QuickDraw uses **RGBColor** structure for referencing color values, which has three similar fields namely red, green and blue. The difference being in range what the individual color component can take. RGBColor's components are defined as short giving then a range of 0x0000-0xFFFF. To handle this situation, keep conversion routines ready for converting RGBColor to COLORREF and vice-versa (by up/down scaling the values by a factor of 0xFF)

- ***BitBlt/StretchBit vs. Copybits***

QuickDraw offers **CopyBits** API for transferring the graphics content from one **Gworld** to another. The corresponding functions available on Windows are **BitBlt**, **StretchBlt**. The difference being that CopyBits function has more to offer than its Windows counterpart does. CopyBits function can be used to copy bits from source to destination with or without scaling (you just provide different source and destination rectangles for desired scaling), whereas on Windows you have two different functions for them. BitBlt can be used to transfer the bits from source to destination device context, but it doesn't support scaling. For scaling the image while transferring, you need to use StretchBlt API.

However, the main problem is faced while handling the bitmap with transparency. CopyBits can be used to draw with transparency also, and all you need to do is to specify the drawing mode to be transparent as one of its parameter and CopyBits automatically considers the current background color as the transparent color. On Windows, drawing with transparency can be a little tricky (though Win32 API has call **TransparentBlt** for same operation, it is not supported for Win95 or WinNT). Handling of transparent bits using BitBlt/StretchBlt requires a two-step process involving an intermediate memory device context and using different drawing modes.

Design Time Availability On PowerPlant vs. Runtime Coding On Windows

- ***Restricting Windows Minimum Width/Height***

Many a time it happens that due to some reason you don't want to allow users to resize your windows beyond some minimum size. *PowerPlant Constructor* (PP's resource editor) has design time option for specifying windows minimum width and height. On windows, there is no such option at design, rather you have to explicitly capture the WM_GETMINMAXINFO message and set the minimum size allowed at the run time.

- ***Scrolling Views***

PP constructor has an option for specifying the scrolling view Id for a **LScrollingView** at the design time, and it automatically handles the scrolling of the contained view. You can also specify the scrollbar position and width along with visibility state for horizontal and vertical scroll bars individually. This translates into a lot of coding to be done on Windows by trapping scrollbar notification messages.

- ***Buttons with Attached Pictures/Popup Menus***

Some of the button classes used by PP (**LPushButton, LBevelButton** etc.) have options for attaching icons/picture to them. LBevelButton can also have a popup menu attached to it. Attaching an icon, a picture or a menu to LBevelButton is as easy as specifying the type and resource ID of the items to be used. This feature handling has to be accommodated into the code on Windows.

Conclusion

The differences in Mac and Windows operating system, in terms of the native technologies and the development tools used on both the platform, adds enough thrill to the porting process of an application from one platform to another. Sometime you are up front with issues, which are hard to solve on the target platforms and are unique enough to be surfaced in your situation only. Barring those hard-lucks, if you are porting an application from Mac to Windows, reading this paper would have at least made sure that you don't spend your time in re-dealing the problems already discussed here.

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

---