POWERPLANT WORKSHOP

By Aaron Montgomery

Basic Windows

How one goes about writing a PowerPlant application

About the author...

Aaron teaches in the Mathematics Department at Central Washington University in Ellensburg, WA. Outside of his job, he spends time riding his mountain bike, watching movies and entertaining his wife and two sons. You can email him at `montgoaa@cwu.edu`, try to catch his attention in the newsgroup `comp.sys.mac.oop.powerplant` or visit his web site at `mac69108.math.cwu.edu:8080/`.

THESE ARTICLES

This is the third article in a series of articles about Metrowerks' PowerPlant application framework. The first article introduced how the framework deals with commands and the second article discussed the debugging facilities of the framework. This article focuses on the window classes in the framework. This series assumes familiarity with the C++ language, the Macintosh Toolbox API and the CodeWarrior IDE. The articles were written using CodeWarrior 6 with the net-update to IDE 4.1.0.3 and no other modifications. Throughout this and future articles, I will assume that you are using the class browser and the debugger to explore the PowerPlant code more fully.

PANES, VIEWS AND WINDOWS (OH MY)

Windows provide your users the ability to view their data and so they will be a fundamental part of your user interface. PowerPlant helps you in providing this visual interface through its `LPane`, `LView` and `LWindow` classes. An `LPane` is something that is drawn on the screen. An `LView` is an `LPane` that may contain other `LPane`s (which allows you to design a visual hierarchy). An `LWindow` is usually the topmost `LView` in a visual hierarchy and it is derived from both `LView` and `LCommander`. **The PowerPlant Book** spends multiple chapters on these classes so you can (correctly) assume that I will leave a lot unsaid. The purpose of this article is to introduce you to Constructor and some of the basic code needed to work with the visual hierarchy in PowerPlant.

CONSTRUCTING WINDOWS

My general pattern of development is to first set up the visual hierarchy in Constructor and then write the code to support that hierarchy. This ordering is not required and I frequently go back and forth between Constructor and the CodeWarrior IDE. For the sake of keeping the article short, we will inspect the `AppResources.ppob` file first and the source code second. The application uses three windows: a document window for text editing, a dialog that requests information from the user and a floating window for displaying information to the user. We will examine the document window first.

Double-clicking the `AppResources.ppob` in the project window should launch Constructor. You can do this resource editing in Resourcerer (using templates provided) or in ResEdit (if you are really got at hex), but I have always found Constructor more convenient. The discussion here is limited (remember, this is an introduction to PowerPlant, not a reference) but you can find more information in the **Constructor User Guide** on the CodeWarrior CD. Double-click the "Document Window" resource (you may need to expand the **Windows & Views** group to see it) and select **Show Object Hierarchy** from the **Layout** menu. A document window has three nested views: an `LWindow` object containing an `LScrollerView` object containing an `LTextEditView` object (well, not quite, but we will get to that in a moment). Open a **Property Inspector** window (from the **Windows** menu) and select the `LWindow` from the **Hierarchy** window. Most of the properties should be familiar to you if you have worked with Macintosh window. You can learn about many of the ones that aren't familiar by adjusting them and then running the application. The first property is the Class ID that is used by PowerPlant to determine exactly what sits in each resource, we will explain it later, but for now you will want to leave it untouched.

Inside the `LWindow` is an `LScrollerView` that handles the scrollbars for the window. Click on it in the **Hierarchy** window and look at its properties. `LScrollerView` is derived from `LPane` and the first set of values is used by the `LPane` base class. The Pane ID is necessary if you need to have access to this pane from your source code, since I don't access the scrollbars from the source code I'll leave it set to 0. Just like the `LWindow` resource, don't change the Class ID number. The `LScrollerView` is 2 pixels higher and wider than the enclosing window (and placed one so that it extends 1 pixel beyond each edge). I can never remember the appropriate values for the Height and Width properties so I usually dig around in the PowerPlant stationary or PowerPlant examples and examine the values there. This `LScrollerView` is bound to all four sides (so it will expand in all directions if the window expands). The second set of properties is specific to the `LScrollerView` class. The first is the thickness of the scrollbars as well as their positioning. Again, I don't typically remember these off the top of my head and so I end up looking in stationary and example files. By setting the horizontal scrollbar's Left and Right properties to -1 we cause it to be absent. The vertical bar is set to 0 from the top and 15 from the bottom (more values gleamed from other sample resources). The Scrolling View ID is the Pane ID of the pane that the scrollbars will control (in this case, it is the `LTextEditView`). I have activated Live Scrolling.

Inside the `LScrollerView` is something based on an `LTextEditView`. If you look at an untouched `LTextEditView`, the Class ID is `txtv`, but this has a Class ID of `Htxv`. We are not going to be using a `LTextEditView` in our application, but rather a subclass called `CHTMLTextView`. Since the subclass doesn't need any more data than what is in a `LTextEditView` resource, the easiest way to create a resource for a `CHTMLTextView` is to use the `LTextEditView` template and then change the Class ID. The effects of this change will be discussed below when we talk about the source code. I've discussed the `LPane` properties in the last paragraph. Since I will try to access this pane from the source code, I needed to give it an "honest" Pane ID and I choose `Text` since this is the text view. The choice to give the `LTextEditView` a 2 pixel edge is again based on looking at various samples. The `LView` properties are left untouched (and are explained in **The PowerPlant Book**).

The `LTextEditView` properties contain `TextEdit` flags that affect the Macintosh `TextEdit` structure. I have set Word Wrap on (since I have eliminated the horizontal scrollbar, this is important). The `TextTraits` is a resource ID and checking the appropriate resource in Constructor shows that these traits set the font size to 12 and the font to Monaco (since it is mono-spaced). The TEXT resource ID is the location of the original text in the window (the TEXT resource is in the file `AppResource.rsrc` since I use Resourcerer to edit it). In some later article, I might talk about how to create templates so that you can add data to be edited in Constructor for your classes.

Before looking at the other two windows, a natural question might arise: where did these I get these classes to put into the hierarchy? You can create a new window resource by dragging an item from the **Windows** tab of the **Catalog** window (choose **Catalog** from the **Window** menu) into your resource file's window. Then you open the window you just created and build the hierarchy within this resource by dragging items from the **Catalog** Window into the resource's window. You should probably spend some time just browsing your options. One thing to notice is that some visual features appear more than once. In the **Views** tab, you will find an `LScroller`, an `LActiveScroller` and an `LScrollerView`. The first two pre-date appearance classes but I cannot seem to find any PowerPlant documentation that helps you make this determination. If you add these to your window and then try to register them in your source code (a step described below), you will be required to add files to your project. This is an indication that a newer class is available. Other examples are the `LRadioGroup` in the **Other** tab (replaced by the `LRadioGroupView` in the **Views** tab) and `LEditField` in the **Panes** tab (replaced by `LEditText` in the **Appearance** tab). The easiest way to determine what to use is to dig around in the sample ppob files provided by Metrowerks.

The next window, Glossary Window, is a floating window. The actual steps in constructing it are not very different from the steps required in the previous example. The only things I will point out are that the `LTextEditView` is set to be neither Selectable nor Editable.

The final window is a dialog box. This is created by dragging an `LGADialog` from the **Catalog** window to your resource file's window. You can then create the visual hierarchy in the same manner as for regular windows. You will need to set the Default Button ID and the Cancel Button ID. This will allow your dialog to correctly handle user requests from the keyboard. When you actually create those buttons, you will set their messages to `msg_OK` and `msg_Cancel` and check "Is Default Button" for the OK button as well. We will discuss the messaging mechanism in PowerPlant in more depth in the fifth article, if you cannot wait, look in **The PowerPlant Book**.

The only other changes are the removal of the **Bad Things** menu, the addition of two commands to the **Edit** menu and the addition of an **Insert** and a **Glossary** menu. You should be able to determine how these were created and what they do from the first and second articles so I will move on to discussing the changes to the source code.

Checking that the class hierarchy has been constructed correctly can be done using the **Visual Hierarchy** menu item in the **Debug** menu. This will present a floating window displaying all of the hierarchies of the front-most regular application windows. You can even identify panes by moving the mouse over them and watching the pane information turn green in the **Visual Hierarchy** window.

CHANGES TO EARLIER CLASSES

Just as PowerPlant has two implementation files for the `UDebugging.h` header, there are three implementation files for the `UDesktop.h` header. The basic implementation is

UDesktop.cp, UFloatingDesktop.cp should be used if you use floating windows. Finally, UWMgr20Desktop.cp should be used if you know that the code will be run using version 2.0 of the Window Manager. This project uses UFloatingDesktop.cp because it does contain floating windows, but does not require any of the UWMgr20Desktop.cp implementations.

There are no code changes in the main() function so we turn our attention to the CDocumentApp class. We remove all the **Bad Things** code from the previous article and add the Lookup command (which is always enabled).

This moves us to the CTextDocument class. Although very little of the code is new, we will explain some of the existing lines of code for the first time. We start with this classes constructor:

CTextDocument.cp

```
CTextDocument::CTextDocument(
  LCommander*    inSuper)
  : LSingleDoc(inSuper)
{
  mWindow = LWindow::CreateWindow(PPob_TextWindow, this );
  ValidateObject_(mWindow);

  myTextView =
    FindPaneByID_(mWindow, kTextView, CHTMLTextView);
  ValidateObject_(myTextView);
  mWindow->SetLatentSub(myTextView);

  NameNewDoc();

  mWindow->Show();
}
```

The LSingleDoc class (CTextDocument's superclass) has an LWindow* member data named mWindow. We use this store a pointer to the document's window. The call to CreateWindow() takes a resource id as well as a commander. In the first article, you learned that the commander will be the window's supercommander. Here we discuss how PowerPlant builds the LWindow from the resource. The actual conversion from resource data to class object is done by the PowerPlant classes called UReanimator and URegistrar. PowerPlant opens the resource as an LStream. Then PowerPlant reads the first four characters of the resource and looks these up in a static table that associates four-character codes to class constructors taking an LStream as input. If the four-character code is found, then the associated constructor is used to create the object. If the four-character code is not found, an exception is thrown. You can witness such an event if you replace the Class ID of the CHTMLTextView object in the Document Window with HTXT and then run the application. The natural question now is how PowerPlant knows which constructors are associated with which Class IDs. The static table used is generated by calls to the macro RegisterClass_(). This will associate the class's class_ID four-character code with the class's stream constructor. You will need to make sure all of your PowerPlant resources are properly registered before you attempt to read them in from the file. You can use the **Validate PPob…** and **Validate All PPobs** menu items from the **Debug** menu to verify that you have made all of the necessary registrations.

Once we have a pointer to the window, a natural thing to do will be to try to access the things in the window. In this particular instance, we will want to establish the text view as the latent subcommander for the window (meaning that when the window is placed on-duty, the text view takes control). You can access the subpanes of a window using the `FindPaneByID_()` macro. This macro takes the window, the Pane ID and the type of that pane as its input. If you access a pane with this macro, you will need to make sure that the pane has a unique Pane ID. The `FindPaneByID_()` will attempt to find the subpane using the given information and cast it to the appropriate type (throwing an exception if either of these operations fail). Now that we have a pointer to the appropriate commander, we can establish it as the latent subcommander.

CHTMLTextView

The `CHTMLTextView` class derives from the `LTextEditView` class and adds some features appropriate for editing HTML code. We examine some of the class declaration first.

CHTMLTextView.h

```
class CHTMLTextView
: public PowerPlant::LTextEditView
{
public:
  enum { class_ID = FOUR_CHAR_CODE('Htxv') };

          CHTMLTextView();
          CHTMLTextView(PowerPlant::LStream* inStream);

  //remainder omitted
};
```

The first thing to notice is that the class defines `class_ID` to be `Htxt` so that the `RegisterClass_()` macro will work correctly. You should choose `class_ID`s that contain some uppercase letters (because PowerPlant reserves those which are all lowercase).

Before presenting the code, I'll explain what `CHTMLTextView` does that a regular `LTextEditView` does not do. A `CHTMLTextView` knows how to insert a few of the HTML tags into the document, when it does this insertion, it places a marker (the character •) in places which should be filled out by the user. The user can then use the **Goto Next Marker** and **Goto Previous Marker** items in the **Edit** menu to jump from their current location to the next (or previous) marker. (If you use Alpha as a text editor, then this behavior probably looks familiar). The `CHTMLTextView` is also capable of inserting appropriate HTML tag templates (using the marker for items to be supplied by the user). Some of the code allowing `CHTMLTextView` to handle these requests is below.

InsertTag() in CHTMLTextView.cp

```
void CHTMLTextView::InsertTag(const std::string& inOpen,
  const std::string& inClose)
{
  TEHandle theTextEditH = GetMacTEH();

  StHandleLocker
    theLock(reinterpret_cast<char**>(theTextEditH));
  short theStart = (**theTextEditH).selStart;
  short theEnd = (**theTextEditH).selEnd;

  if (theStart == theEnd)
  {
    string theMarker = "";;
    theMarker += Marker();
    Insert(theMarker.c_str(), theMarker.length());
    theEnd = theStart + 1;
  }

  SetSelectionRange(theEnd, theEnd);
  Insert(inClose.c_str(), inClose.length());
  SetSelectionRange(theStart, theStart);
  Insert(inOpen.c_str(), inOpen.length());
  SetSelectionRange(theStart, theStart);
}
```

We obtain the `TEHandle` directly with a call to `GetMacTEH()` and then use a `StHandleLocker` to lock the handle until the `StHandleLocker`'s destructor is called. Although it is unlikely that the lock is currently needed and so the lock is unnecessary, that might make the code fragile if the PowerPlant code changed. Furthermore, because the lock is controlled by an object on the stack, I feel confident that the lock will be released. We obtain the starting and ending of the selection directly from the `TEHandle`. If no item is selected, we insert a marker into the text and make it the current selection. Then we insert the tag close and tag open and place the cursor at the start of the tag. The code needed to go to the next marker has a similar flavor.

GotoNextMarker() in CHTMLTextView.cp

```
void CHTMLTextView::GotoNextMarker()
{
  TEHandle theTextEditH = GetMacTEH();

  StHandleLocker
    theLock1(reinterpret_cast<char**>(theTextEditH));

  short theEnd = (**theTextEditH).selEnd;
  short theLength = (**theTextEditH).teLength;
  Handle theTextH = (**theTextEditH).hText;

  StHandleLocker theLock2(theTextH);

  for (short thePosition = theEnd;
    thePosition != theEnd - 1;
    ++thePosition)
  {
    if (thePosition > theLength)
    {
      thePosition = 0;
    }
    if ( *((*theTextH) + thePosition ) == Marker() )
    {
      SetSelectionRange(thePosition, thePosition + 1);
      return;
    }
  }
}
```

Both of these panes use PowerPlant's code to handle drawing and user inter-action. If you need to override these (and other) routines, you need to be aware of how PowerPlant factors the work. There is both a `Draw()` and a `DrawSelf()` method, similarly, there is both a `Click()` and a `ClickSelf()` method. You should override the `DrawSelf()` method and call `Draw()` method in your source. The `Draw()` method will do the necessary set up, then call the `DrawSelf()` method and then do the necessary tear down.

CGLOSSARY

The `CGlossary` class provides a first glimpse into how dialogs are handled in PowerPlant (they will be discussed more in depth in the fifth article). The first thing to notice is that `CGlossary`'s constructor is private and unimplemented. This is because the class currently has no data and only static methods. The class could have been implemented as a namespace at this point, but it has been written as a class to allow for the possibility of future improvements (for example, storing the glossary data in an associative array in RAM). The class only has two methods: `RegisterClasses()` and `Lookup()`. `RegisterClasses()` does the usual task of registering those visual classes which are used by the `CGlossary` class and is not presented here. The `Lookup()` method is overloaded with two implementations. The first takes no arguments, presents the user with a dialog asking the tag to look up, and then invokes the second version. The second version takes a string argument and presents the user with the tag's glossary information (if it exists). Their code is presented below.

Lookup() from CGlossary.cp

```
void CGlossary::Lookup()
{
  unsigned char theString[256];
  theString[0] = '\0';

  StDialogHandler  theHandler(k_PPob_GlossaryDialog,
    LCommander::GetTopCommander());
  LWindow*   theDialog = theHandler.GetDialog();

  LEditText*   theField
    = FindPaneByID_(theDialog,
        k_PaneIDT_Lookup, LEditText);

  theField->SetDescriptor(StringLiteral_(""));
  theField->SelectAll();
  theDialog->SetLatentSub(theField);
  theDialog->Show();

  while (true)
  {
    MessageT hitMessage = theHandler.DoDialog();

    if (hitMessage == msg_Cancel)
    {
      return;
    }
    else if (hitMessage == msg_OK)
    {
      theField->GetDescriptor(theString);
      break;
    }
  }

  Lookup(theString);
}
```

We use a `StDialogHandler` to manage the dialog that obtains user input (more on this in the fifth article). We pass in the resource ID of the appropriate dialog and obtain the dialog window from the Handler. We use `FindPaneByID_()` to find the `LEditText` in that window and present the window to the user. We then enter a loop calling the `DoDialog()` method of `theHandler` repeatedly. This will return information about what items were hit in the dialog. If the **Cancel** button was hit, we simply leave the function. If the **OK** button was hit, we obtain the user's input using the `GetDescriptor()` method of the `LEditText` and break from the loop. The handler ignores any other action by the user. Once the dialog has been dismissed we call the other `Lookup()` method (because the **OK** button was hit).

Lookup() in CGlossary.cp

```
void CGlossary::Lookup(Str255 inTerm)
{
  StResource theDefinition;

  try
  {
    theDefinition.GetResource(k_ResType_TEXT, inTerm,
      true, true);
  }
  catch (...)
  {
    theDefinition.GetResource(k_ResType_TEXT,
      k_Str255_Unknown, true, true);
  }

  LWindow* theWindowP
    = LWindow::CreateWindow(k_PPob_GlossaryWindow,
        LCommander::GetTopCommander());
  LTextEditView* theDefnP
    = FindPaneByID_(theWindowP, k_PaneIDT_Definition,
        LTextEditView);
  theDefnP->SetTextHandle(theDefinition);

  LStaticText* theTermP
    = FindPaneByID_(theWindowP, k_PaneIDT_Term,
        LStaticText);
  LStr255  theString = StringLiteral_("<");
  theString += inTerm;
  theString += StringLiteral_(">");
  theTermP->SetDescriptor(theString);
}
```

The second version of `Lookup()` is responsible for actually determining and displaying information about the tag. The information is stored as resources of type `TEXT` with the name of the tag used as the name of the resource. We obtain the resource using a `StResource` that will dispose of the resource when its destructor is called. We first try to obtain a resource with the tag's name and if that fails, we use a generic error message stored in a resource with the name "Unknown". The first `true` in the `GetResource()` call tells the method to `Throw_` if there is an error and the second `true` tells the method to only look in the current resource file.

Once we have the text we create a window, use some `FindPaneByID_()`'s to obtain pointers to some text fields and fill them in. We use the `LStr255` class from PowerPlant for the second field. This class is described in a *MacTech* article written by John C. Daub and provides a lot of the functionality normally found in the C++ string class for Pascal style strings (of length up to 255). The `StringLiteral_` macro converts its input into Pascal style strings. Currently it is very simple, but using the macro will make it easier to adjust if the PowerPlant API changes to C style strings at some future point.

CONCLUDING REMARKS

Well, that provides a user interface to your application. Although it was quick and sketchy, it should give you some feel for how PowerPlant handles windows. We explore how PowerPlant works with files in the next article (which is the other half of PowerPlant's `LDocument` class). The core portion of this series will be completed in the fifth article where we cover how PowerPlant handles dialogs. Once the core portion is completed, I am open for suggestions on where to go next, so if you have any ideas, please e-mail me.

POWERPLANT REFERENCES

References that are particularly appropriate for this article are the following:

"Panes", "Views", "Windows" chapters in **The PowerPlant Book**

John C. Daub's "The Ultra-Groovy LString Class" in *MacTech*, January 1999