POWERPLANT WORKSHOP

By Aaron Montgomery

Basic Dialogs

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 fifth article in a series of articles about Metrowerks' PowerPlant application framework. The first article introduced how the framework deals with commands, the second article discussed the debugging facilities of the framework, the third introduced windows, and the fourth article tacked file classes. This article completes the discussion of the core of PowerPlant by discussing the way PowerPlant handles dialogs. 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.

BROADCASTERS AND LISTENERS

We have talked about the command chain that PowerPlant uses to convert menu selections to code calls, but trying to pass all of the user interaction with a dialog box through this chain would be cumbersome. Fortunately, PowerPlant offers an alternative method of linking different classes together through its `LBroadcaster` and `LListener` classes. Each `LBroadcaster` object keeps a list of `LListener`s and can broadcast a message to the objects in its list. The broadcast is done by calling each `LListener`'s `ListenToMessage()` method. This method takes two parameters. The first, of type `MessageT`, describes the message being broadcast. The second, of type `void*,` allows the `LBroadcaster` object to pass message specific data to the `LListener` object.

Although this article focuses dialogs, the `LBroadcaster` and `LListener` classes can also be used in other places. For example, the `LGrowZone` class (mentioned in the second article) broadcasts a message when it needs more memory. If you have a class that can release memory in a pinch, you might want to make it an `LListener` and register it with the `LGrowZone` object.

In the case of dialogs, the control classes are LBroadcasters and will broadcast their message whenever they are adjusted. This means that if you want something to happen when a control is adjusted, you can create an LListener and have it listen to that control. This means that much of the code for dynamic effects in dialogs can be placed in separate classes and linked to the dialog before it is presented to the user.

CONSTRUCTOR

Before looking at the resource file, you may want to run HTMLedit and play with the controls in the dialog (select **Preferences…** from the **Edit** menu). Now go to the IDE and double-click the AppResources.ppob file (which should open in Constructor).

The Preferences dialog has two LPushButtons (**OK** and **Cancel**), an LMultiPanelView (where everything important happens) and an LTabsControl (to switch between the panels). The two LPushButtons did not require much set up, but you need to remember to let the LGADialog know the button IDs for the default and cancel buttons. The LTabsControl needs to have a valid **Value Message**. I set the value to SwPa for Switch Panel. This is the message that will be broadcast when someone clicks on the control. The LTabsControl needs to have an associated tab# resource which determines the titles of the tabs (as well as icons, if any are used on the tabs). Unfortunately, I do not think the tab# resource can be edited in Constructor. You have two choices: add it to the AppResources.rsrc file or open the AppResources.ppob file in Resourcerer or ResEdit. If you choose to keep the tab# resource in the AppResources.rsrc file, you will not see the tab names in Constructor. Because I like to see these while working on the dialog, I added the tab# resource to the AppResources.ppob file with Resourcerer.

Now to the centerpiece of the dialog: the LMultiPanelView. To add panels to a LMultiPanelView in Constructor, you select the **Panels** cell in the **Property Inspector** window and then select **New Panel** from the **Edit** menu. In this case, there are three (one for each tab). You then select which PPob resource to associate with each panel. In our case, we use resources with IDs 10031, 10032 and 10033. The final settings which need to be made for this resource are the setting of the **Switch Message** (it should match with the LTabsControl) and selecting the **Listen To SuperView** checkbox. The LMultiPanelView is an LListener and when it hears the Switch Message, it will switch to another panel. Selecting the **Listen To SuperView** checkbox adds the LMultiPanelView object as a listener to the LTabsControl object.

Now it is time to construct the three views that will appear in the LMultiPanelView. The first is used for determining the file creator of saved files. This LView was created by dragging an LView from the **Catalog** window into the AppResources.ppob window and then changing its Resource ID to 10031. The size of the panel needs to be adjusted to fit inside the LMultiPanelView. The **Visible** checkbox needs to be deselected (otherwise, all of the LViews will appear at once in the LMultiPanelView). The primary control here is a LRadioGroupView and four LRadioButtons. Since there is no need for the buttons to broadcast a message when they are hit, they do not need **Value Message**s. However, since the code will need to know which button is selected, the **Pane ID**s need to be set. The **Pane ID**s match the appropriate creator code for each application. Notice that there is an older

LRadioGroup in the **Other** tab of the catalog. This has been replaced by the LRadioGroupView in the **Views** tab.

The second LView (Resource ID 10032) used in the LMultiPanelView is very simple and straightforward. One way to save a little work is to duplicate the LView with Resource ID 10031, renumbered it to 10032 and then deleted all of the controls. This means that the new resource will be the correct size and visibility. This LView has an LStaticText and an LEditText. The LEditText is given a **Pane ID** because we will need to be able to access its contents from the code. The LEditText is from the **Appearance** tab and replaces the LEditField in the **Panes** tab.

The third LView (Resource ID 10033) is the most complicated. Again, creation is done by duplicating the LView with Resource ID 10031, changing the Resource ID, deleting all the existing controls and adding new ones. It has an LPopupButton and an LMultiPanelView. The **Message Value** for the LPopupButton is SwPa for Switch Panel. The **MENU Resource ID** is set to 1002 (you can find this resource in the AppResources.ppob file as well). The LMultiPanelView uses Resource IDs 10034 and 10035. The **Switch Message** is set to SwPa so that it is consistent with the LPopupButton. Unlike the first LMultiPanelView, this one cannot simply listen to its SuperView and we will examine the code needed to link the LPopupButton to the LMultiPanelView below. Resources 10034 and 10035 are very basic, consisting of some LCheckboxes and LStaticTexts. The LCheckboxes have had their **Message Value**s set because the code will need to listen for their broadcasts.

This concludes our tour of the AppResources.ppob file and we now turn our attention to the source code. As you build your interface, if you cannot find a control by browsing through the **Catalog** window, ask on the comp.sys.mac.oop.powerplant newsgroup. Usually someone will be able to tell you where it is in the **Catalog**, suggest a class from the PowerPlant Archive (at Metrowerks' web site), or provide a work around.

PREFERENCE CLASS

The CPreferences class handles the application's preferences. The class consists of a constructor; a destructor; a method to register used classes; a method to set the preferences via a dialog; and a method to alert other classes about a change in preferences. We begin with the constructor.

CPreferences() in CPreferences.cp

```
CPreferences::CPreferences()
: myFile(StringLiteral_("HTMLedit Prefs"))
{
  myFile.OpenOrCreateResourceFork(fsRdWrPerm
      OSType_Pref_Creator,
      OSType_Pref_FileType,
      smSystemScript);
  {
    StNewResource thePrefs(ResType_Pref,
                    ResIDT_Pref,
                    sizeof(SPreferences));
    if(!thePrefs.ResourceExisted())
    {
      SPreferences* thePrefsP
                    = reinterpret_cast<SPreferences*>(
                            *thePrefs.mResourceH);
      thePrefsP->FileType = OSType_Default_Creator;
      thePrefsP->Marker = char_Default_Marker;
      thePrefsP->LinkUsesTarget
                = bool_Default_LinkUsesTarget;
      thePrefsP->ImageUsesAlt = bool_Default_ImageUsesAlt;
      thePrefsP->ImageUsesBorder
                = bool_Default_ImageUsesBorder;

      thePrefs.SetResAttrs(resLocked + resPreload);
    }
  }

  myPrefsH = reinterpret_cast<SPreferences**>(
                ::Get1Resource(ResType_Pref, ResIDT_Pref));
  if (myPrefsH == nil)
  {
    ThrowIfResError_();
    ThrowIfNil_(myPrefsH);
  }

  Update();
}
```

The data member `myFile` is of type `LPreferencesFile`. This class is provided by PowerPlant to create preference files in the `System` folder. In this case, the code names the preference file `HTMLedit Prefs` and then opens the resource fork (creating it if needed).

The `StNewResource` class will attempt to open the specified resource and if it does not exist, it will create the resource. The `ResourceExisted()` method of that class will return `false` if the resource is new and in this case, the code fills the resource with the default preferences. The code also sets the resource attributes to preload and lock the resource since we will want to have it available at all times. The resource structure is small and so it is unlikely to cause an undue memory burden here. When the `StNewResource` passes out of scope, the resource will be written into the preference file.

Next the code loads the resource and then passes all of the preferences to those classes using them. I've decided to use a raw resource as a data member instead of using a `StResource`. This means that the code needs to call `::ReleaseResource()` in `CPreferences`' destructor. I felt that there was no need to use a stack class to handle memory management because the resource will be loaded until the application quits. If you are going to add a resource data member to a class in other situations, you should consider using the `StResource` class instead of a raw resource.

The most interesting method in the `CPreferences` class is the `Set()` method which runs the dialog. We present the code for this method now.

Set() from CPreferences.cp

```
void CDocumentApp::CPreferences::Set()
{
  StDialogHandler    theHandler(PPob_Prefs,
                         LCommander::GetTopCommander());
  LWindow*     theWindowP = theHandler.GetDialog();

  LMultiPanelView* theMPViewP = FindPaneByID_(theWindowP,
                                PPob_Prefs_MPV,
                                LMultiPanelView);
  theMPViewP->CreateAllPanels();

  LRadioGroupView* theRadioGroupP =
      FindPaneByID_(theWindowP,
        PPob_Prefs_FileType, LRadioGroupView);

  theRadioGroupP->SetCurrentRadioID((**myPrefsH).FileType);

  LEditText*     theFieldP = FindPaneByID_(theWindowP,
                             PPob_Prefs_Marker,
                             LEditText);
  LStr255        theMarker = (**myPrefsH).Marker;
  theFieldP->SetDescriptor(theMarker);

  theMPViewP = FindPaneByID_(theWindowP,
                   PPob_Prefs_MPV2, LMultiPanelView);
  theMPViewP->CreateAllPanels();

  LPopupButton*    thePopupP = FindPaneByID_(theWindowP,
                                PPob_Prefs_Popup,
                                LPopupButton);
  thePopupP->AddListener(theMPViewP);

  LCheckBox*     theCheckBoxP = FindPaneByID_(theWindowP,
                                  PPob_Prefs_LinkTarget,
                                  LCheckBox);
  theCheckBoxP->SetValue((**myPrefsH).LinkUsesTarget);
  LStaticText*   theStaticTextP = FindPaneByID_(theWindowP,
                                  PPob_Prefs_LinkTemplate,
                                  LStaticText);

  CDynamicLinkText theDynamicLinkText(theStaticTextP,
                          theCheckBoxP);
```

//omitted code setting up other panel

```
  theWindowP->Show();

  while (true)
  {
    MessageT theMessage = theHandler.DoDialog();
    if (theMessage == msg_Cancel)
    {
      return;
    }
    else if (theMessage == msg_OK)
```

```
      {
        break;
      }
    }

    (**myPrefsH).FileType =
        theRadioGroupP->GetCurrentRadioID();

    theFieldP = FindPaneByID_(theWindowP,
                    PPob_Prefs_Marker, LEditText);
    theFieldP->GetDescriptor(theMarker);
    (**myPrefsH).Marker = theMarker[1];
```

//omitted code updating preferences handle

```
    ::ChangedResource(reinterpret_cast<Handle>(myPrefsH));
    ::UpdateResFile(myFile.GetResourceForkRefNum());

    Update();
}
```

Although it looks like a lot of code, there are only four basic steps. First, set the values of the controls based on the current preferences. Second, set up the dynamic text. Third, run the dialog box. Fourth, update the preferences based on the user interaction.

The `StDialogHandler` class was introduced in the third article and one is used here. The first parameter is the Resource ID of the dialog and the second parameter is the super commander of the dialog. Since we will need to adjust some of the user interface before presenting the dialog to the user, we use the `GetDialog()` method to obtain the dialog window. We now look at the interesting bits of the code.

The call to `CreateAllPanels()` is important because we will want to access the panels prior to user interaction. If you do not call this, the panels of the `LMultiPanelView` will be created as the user switches from one panel to another. If the panels were completely static, this approach would be appropriate. Because we are going to adjust each panel before presenting it to the user, it is easiest to adjust everything before starting to interact with the user.

The next call that is new is the call to `AddListener()`. This call links the `LPopupButton` in the third panel to the `LMultiPanelView` in the third panel. Both the `LTabsControl` class and the `LPopupButton` class pass the tab or item number chosen as the `void*` parameter in their call to `ListenToMessage()`. This is exactly what the `LMultiPanelView` class expects and so there is no need for any additional code to make the `LMultiPanelView` objects work.

The line creating a `CDynamicLinkText` object will be explained below. What is convenient is that in this code, simply the creation of these objects is all that is necessary to generate dynamic effects in the dialog. The actual code to do that work does not need to clutter up the code handling the user interaction with the dialog.

Next we show the window and let the `StDialogHandler` object run the dialog. The return value from `DoDialog()` will be the last message broadcast by an `LBroadcaster` in the dialog box (the `StDialogHandler` is an `LListener` that listens to the controls in the dialog). The only two messages we need to concern ourselves with are the `msg_Cancel` and msg_OK, all the other messages are handled by other LListener objects. The remainder of the code is straightforward, we obtain the values from the dialog and update the preferences.

We will not discuss the Update() method but its code is straight forward (calling a number of methods to update the user preferences in those classes that use them). We now turn our attention to the CDynamicLinkText and CDynamicImageText classes.

CDynamicLinkText & CDynamicImageText

These two classes are responsible for dynamically updating LStaticText objects in the dialog as the user checks and unchecks LCheckBox controls in the dialog. The two classes are almost identical and we present CDynamicLinkText here. The code is below is presented as a testament to the ease with which these effects can be accomplished with PowerPlant. Other than the code to generate the actual text to be displayed, there are under 10 lines of code in the entire class.

CDynamicLinkText code from CDynamicText.cp

```
CDynamicLinkText::CDynamicLinkText(
      LStaticText* inTextP, LCheckBox* inUseTargetP)
: myTextP(inTextP),
  myUseTargetP(inUseTargetP)
{
  myUseTargetP->AddListener(this);
  SetText();
}

void CDynamicLinkText::ListenToMessage(
      MessageT inMessage, void* ioParam)
{
  ioParam = ioParam;
  if (inMessage == msg_ToggleTarget) { SetText(); }
}

void CDynamicLinkText::SetText()
{
    LStr255 theTag = "";
    theTag += "<A href=\"";
    theTag += '·';
    theTag += "\"";
    if (myUseTargetP->GetValue())
    {
      theTag += " target=\"";
      theTag += '·';
      theTag += "\"";
    }
    theTag += ">";
    theTag += '·';
    theTag += "</A>";
    theTag += '·';

    myTextP->SetDescriptor(theTag);
    myTextP->Refresh();
}
```

`CDynamicLinkText` is an `LListener` and its primary method is the `ListenToMessage()` method. Its constructor accepts a pointer to an `LCheckbox` and a pointer to an `LStaticText`. When the `LCheckbox` object is clicked by the user, it will broadcast a message. The `CDynamicLinkText` object will receive this message and update the `LStaticText` object to reflect the current settings.

OTHER CLASSES

Changes needed to be made to the `CDocumentApp` class, `CHTMLTextView` class and the `CTextDocument` class in order use these preferences. `CDocumentApp` now handles the `cmd_Preferences` (and passes all of the work off to a static `CPreferences` data member). `CHTMLTextView` and `CTextDocument` now hold static data members for holding user preferences as well as methods to set and get this data. If you want to view the actual changes yourself, search the files `CDocumentApp.cp`, `CHTMLTextView.cp` and `CTextDocument.cp` for comments starting with `//•`.

CONCLUDING REMARKS

I haven't covered every useful control that can be placed in a dialog box but should have given you a feel for how to use Constructor to build the interface and the necessary code to run the interface. You may have noticed that this article is somewhat shorter than previous articles. The primary reason is that much of the code for this article has been used in previous articles (in slightly other contexts). At this point, I believe I have presented you with the tools to build a basic Macintosh application using PowerPlant (and covered most of **The PowerPlant Book** topics in the process). I'm planning on taking a month hiatus but returning in October. If you have a topic that you would like to see, please send me an e-mail.

POWERPLANT REFERENCES

For more on using the messaging system built into PowerPlant, you will want to read **The PowerPlant Book** chapter "Controls and Messaging." There is also a "Dialogs" chapter in **The PowerPlant Book** that covers (not surprisingly) dialogs. Another source of information is the source code of PowerPlant as well as the example files. I have found the `Appearance Demo` and the `Grayscale Sample` to be particularly useful as they present a variety of different controls. You can also see more examples of dialogs in the `StdDialogs` demo.