POWERPLANT WORKSHOP

By Aaron Montgomery

Basic Files

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 fourth 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 and the third introduced windows. This article focuses on the file classes in the framework (opening and saving files). 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.

SETTING UP NAVIGATION SERVICES

Before you can use Navigation Services, you need to do some setup. The first step is to modify the common prefix file to let PowerPlant know whether you want to always use the classic file dialogs, require Navigation Services, or use Navigation Services if it is available and the classic dialogs otherwise. This is done in the project's prefix file.

CommonPrefix.h

```
#if PP_Target_Carbon
  #define PP_StdDialogs_Option \
    PP_StdDialogs_NavServicesOnly
  #define  SetTryNavServices_       do { } while (false)
#else
  #define PP_StdDialogs_Option  \
      PP_StdDialogs_Conditional
  #define SetTryNavServices_        \
    PowerPlant::UConditionalDialogs::SetTryNavServices(\
      0x01108000)
#endif
```

For Carbon targets, the code sets the dialogs to be Navigation Services at all times and the `SetTryNavServices_` macro does nothing. For the other targets, the code uses conditional dialogs. The conditional dialogs option will use Navigation Services if it is available and classic dialogs otherwise. The `SetTryNavServices_` macro tells PowerPlant to use Navigation services only if it has version at least 1.1 (PowerPlant stationery explains that earlier versions can cause problems).

The macro `PP_StdDialogs_Option` affects the `UStandardDialogs.h` header by setting the `PowerPlant::StandardDialogs` namespace (abbreviated to `PP_StandardDialogs` in the code) to equal one of `PowerPlant::UNavServicesDialogs`, `PowerPlant::UConditionalDialogs`, or `PowerPlant::UClassicDialogs`. When using the class browser to identify classes and functions, you will often find that the same class or function will appear in each of these three namespaces and you will need to look at the appropriate one.

The other necessary change to the code for Navigation Services is the need to load them at application startup and to unload them at application shutdown. This is done in `CDocumentApp`'s constructor and destructor. In the constructor, the code uses the macro `SetTryNavServices_` to establish what version it requires and then calls the function `PP_StandardDialogs::Load()`. In the destructor, the code calls `PP_StandardDialogs::Unload()`.

As a final touch, I have also added `BNDL`, `open`, and `kind` resources to `AppResources.rsrc`. This provides us with custom icons and the finder with information about what types of files we can open.

DIRTY DOCUMENTS

Before discussing the methods used to open and save files, we will discuss how PowerPlant determines if a file has been modified (in other words, if the document is dirty). The `LDocument` class has an `IsModified()` method which should return `true` if the file has been modified since last read from disk and `false` otherwise. This allows the framework to enable and disable the **Save** and **Revert** menu commands appropriately. In order to make the system work, we will need to use the `SetModified()` functions when the file has been saved to the disk and when it has been modified by the user.

The first change to the code adds an `iAmModified` data member to the `CHTMLTextView` class. The `LTextEditView` class from which it derives provides the virtual method `UserChangedText()` which will be called anytime the user types (or deletes) something from the `LTextEditView`.

UserChangedText() in CHTMLTextView.cp

```
void CHTMLTextView::UserChangedText()
{
    if (IsModified() == false)
    {
            SetUpdateCommandStatus(true);
            SetModified(true);
    }
```

}

The code only needs to do something if this is the first modification. In that case the code tells the menus that they will need to be updated and then sets the object's flag to indicate that it has been modified. If you directly manipulate the text in an `LTextEditView`, the `UserChangedText()` method will not be called automatically. In our case, the `InsertTag()` method needs to indicate that it has modified the document and it does this with a call to `UserChangedText()`. The `IsModified()` and `SetModified()` methods of `CHTMLView` are inline functions which access and set the object's flag.

In the `CTextDocument` class, we change the `IsModified()` method so that it checks its `CHTMLTextView`'s `IsModified()` method before returning a result. In addition, we update the `SetModified()` method so that it also calls the `CHTMLTextView`'s `SetModified()` method. Note that PowerPlant does some housekeeping in `LDocument`'s `SetModified()` method and so we call it from within the new method definition. You could also just copy the code, but that would mean that you would need to update your code if more housekeeping was done within `LDocument`'s method.

OPENING FILES

We are now ready to consider how PowerPlant opens and saves files. The code presented here is modeled on the code in the stationery files provided with PowerPlant. There is one limitation with the strategy employed there that may make it unsuitable for your application. The problem arises because of the need to call `::NavCompleteSave()` if Navigation Services have been used to save the file. The code in the PowerPlant stationery handles this by holding the file open on the disk until the document is closed. This allows the `CTextDocument` object to save the information necessary for the call to `::NavCompleteSave()` until the document is closed. The disadvantage is that the file cannot be manipulated by another application while it is held open by `HTMLedit`. If you need to allow external tools to modify files while your application holds them open, you will need to adjust the code to permit this.

The first change in the code is the removal of some lines that were added in the first article. Now that our application can open files, the lines in `FindCommandStatus()` which disabled the open command can be removed. There is no need to change `ObeyCommand()` since PowerPlant's `LDocApplication` class already has the necessary code. Two `CDocumentApp` methods are necessary for opening files. The first, `ChooseDocument()`, interacts with the user to determine which file should be opened and the second, `OpenDocument()`, opens the document. We start with `ChooseDocument()`.

ChooseDocument in CDocumentApp.cp

```
void
CDocumentApp::ChooseDocument()
{

  LFileChooser theChooser;

  NavDialogOptions*  theOptions =
                       theChooser.GetDialogOptions();
  if (theOptions != 0)
  {
     theOptions->dialogOptionFlags |= kNavSelectAllReadableItem;
  }
```

```
  if (theChooser.AskOpenFile(LFileTypeList()))
  {
    AEDescList   theDocList;
    theChooser.GetFileDescList(theDocList);
    SendAEOpenDocList(theDocList);
  }
}
```

There are really three `LFileChooser` classes, one in each of the namespaces mentioned earlier. At the top of the source file, the code indicates that the one in `PowerPlant::StandardDialogs` should be used. Therefore, the actual class used will be determined by the macros in the prefix file. All three of the `LFileChooser` classes allow you to obtain a pointer to a `NavDialogOptions` structure. The pointer will be 0 if the application is using classic dialogs. If the application is running under Navigation Services, we adjust the flags so that **All Readable Items** is the default choice from the popup menu in the dialog.

Next, the code creates an `LFileTypeList` (the default constructor uses the `open` resource with ID 128). In the case of `HTMLedit`, only `TEXT` files can be opened. If `AskOpenFile()` returns `true`, then the user requested a file be opened, and the code gets the `AEDescList` that describes the file from `theChooser` and sends an Open Apple Event to the application. PowerPlant will convert this Apple Event into a call to `OpenDocument()`. One nice feature is that there was almost no need to write separate code for the three possible situations (Navigation, Conditional, and Classic). Furthermore, you obtain the ability to open multiple files under Navigation Services without doing any extra work: PowerPlant will turn these lists into a sequence of individual open commands. We now turn our attention to the place where the document is actually opened.

OpenDocument() in CDocumentApp.cp

```
void
CDocumentApp::OpenDocument(
  FSSpec*    inMacFSSpec)
{
  LDocument* theDoc =
               LDocument::FindByFileSpec(*inMacFSSpec);

  if (theDoc != 0)
  {
    ValidateObject_(theDoc);
    theDoc->MakeCurrent();
  }
  else
    try
    {
      theDoc = NEW CTextDocument(this, inMacFSSpec);
      ValidateObject_(theDoc);
    }
    catch(LException& theErr)
    {
      if (theErr.GetErrorCode() != noErr)
      {
        throw;
```

```
        }
    }
}
```

First, the code tries to find the document from among the currently open documents using the static `LDocument` method `FindByFileSpec()`. If `FindByFileSpec()` returns a pointer to an `LDocument`, then the code brings it to the front and is finished. If the document is not already open, then the code needs to create a new document and other than the error handling, this is accomplished to `CTextDocument`'s constructor.

The error handling code deserves some mention here. The constructor of `CTextDocument` may throw an exception if the document contains more than 32K of text. While this problem is handled in the `OpenFile()` method of `CTextDocument` (presented below), an exception is the only way to inform the `CDocumentApp` that the file was not opened. My personal convention is to use an `LException` whose error code is `noErr` to indicate that everything has been handled but that the document is invalid. When the exception is thrown, DebugNew will report that the document leaked. If this were a "real" application, I would spend some time determining whether this is a real leak or if DebugNew cannot handle exceptions thrown from constructors. Since this is supposed to be a teaching article, I will leave this task as an exercise for the reader (watch out, Metrowerks pools its memory allocations).

You have now seen all of the significant changes to the `CDocumentApp` class and we turn our attention to the `CTextDocument` class. The `CTextDocument` constructor is modified slightly to accept an optional `FSSpec`. However almost all of the work is passed on to its `OpenFile()` method which is where we will pick up the trail. Just as `LSingleDoc` contains a pointer to an `LWindow` as one of its data members, it also contains a pointer to an `LFile` as another member. This is the real purpose of the `LDocument` class: to tie together the visual presentation (`LWindow`) with the data on the disk (`LFile`). The `LSingleDoc` class assumes that each document uses only one window and one file. Unlike `LWindow` which can be complicated (due to the visual hierarchy), the `LFile` class is simple. It will take care of handling the file reference numbers for both the data and resource forks and most of the methods of the class do exactly what their names indicate.

OpenFile() in CTextDocument.cp

```
void CTextDocument::OpenFile(FSSpec& inFileSpec)
{

  mFile = nil;

  try
  {
    StDeleter<LFile> theFile(NEW LFile(inFileSpec));
    ValidateObject_(theFile.Get());

    theFile->OpenDataFork(fsRdWrPerm);
    StHandleBlock theTextH(theFile->ReadDataFork());
    ValidateHandle_(theTextH.Get());
    ValidateObject_(myTextView);
    myTextView->SetTextHandle(theTextH);
    myTextView->SetModified(false);

    ValidateObject_(mWindow);
    mWindow->SetDescriptor(inFileSpec.name);
```

```
    mIsSpecified = true;

    mFile = theFile.Release();
  }
  catch (LException& theErr)
  {
    if (theErr.GetErrorCode() == err_32kLimit)
    {
      UModalAlerts::StopAlert(ALRT_BigFile);
      throw LException(noErr);
    }
    else
    {
      throw;
    }
  }
}
```

The first goal of `OpenFile()` is to provide an `LFile` for the `CTextDocument` object. The code starts by creating a pointer to an `LFile`. The `StDeleter` class is analogous to the standard library's `auto_ptr` template and it is used here so that we do not leak the `LFile` pointer if an exception is thrown. The call to the `StDeleter`'s `Get()` method returns the actual pointer and the code verifies that it is valid.

The code then opens the data fork of the `LFile`, copies the text into a handle (the `StHandleBlock` guarantees the memory is deleted at the end of the scope), and passes the handle to the document's `CHTMLTextView` object. Since the data in the `CHTMLTextView` is fresh from the disk, the code indicates that the `CHTMLView` is unmodified.

The code then sets the title of the window to match the name of the file. The `mIsSpecified` data member of the `LDocument` class indicates that this document has a file associated with it (important since it determines if the **Revert** command is valid). If everything went well, the code sets `mFile`. The call to `Release()` causes the `StDeleter` to disown the `LFile` pointer, failing to do this will cause DebugNew to (correctly) report a double-delete (once when the `StDeleter` goes out of scope and later in the `CTextDocument`'s destructor).

The call to `SetTextHandle()` will throw an exception if the handle contains more than 32k characters. Since we know how to handle this problem here, we do so. The `UModalAlerts` method simply displays an alert stating that the file was too big (you need to supply the resource and PowerPlant does the rest). Since the code needs to let the calling function know that the file was not opened and there is no return value, I have opted to throw an `LException` with error code `noErr`. By personal convention, this means that something bad has happened but that no further remedies are needed. The first place where this error can be ignored should catch the exception and ignore it (you saw this in the `OpenDocument()` method of `CDocumentApp`). That completes the tour of the code necessary to open files using PowerPlant.

Although reversion of a document to the data on the disk is not essential, it is easy to add this ability to a PowerPlant application. PowerPlant stationery assumes that you will do this

since it provides the **Revert** command in its **File** menu and adding the code usually adds little work.

DoRevert() in CTextDocument.cp

```
void CTextDocument::DoRevert()
{
  ValidateObject_(mFile);
  StHandleBlock  theTextH(mFile->ReadDataFork());
  ValidateHandle_(theTextH.Get());

  ValidateObject_(myTextView);
  myTextView->SetTextHandle(theTextH);

  SetModified(false);

  myTextView->Refresh();
}
```

The code reads the data from the `CTextDocument`'s `mFile` into a handle (using a `StHandleBlock` to prevent a leak). Then it updates the text in the `CHTMLTextView`. The code then indicates that the document is now clean. The last step is to make a call to `Refresh()`. One important thing to be careful with here is that the call to `Refresh()` resolves into a call to `::InvalPortRect()`. This means that no actual drawing will be done until the next Update event is handled. Therefore, if you have stopped the event processing queue, then calls to `Refresh()` will appear to do nothing.

Although I will not discuss it in this article, I have also made the `NameNewDocument()` method a little more sophisticated. In retrospect, this really should have been done in the third article where we discussed windows. You might want to examine the code on your own. Now we turn to saving (which is a little more complicated).

SAVING FILES

All saving is handled by the `CTextDocument` class (and not by the `CDocumentApp` class). Three methods need to be implemented for PowerPlant to handle saving files. The first method, `AskSaveAs()`, presents the user with a dialog to determine where to save the file. The second method, `DoAESave()`, method implements as Save As operation (which handles the Save Apple Event). The third method, `DoSave()`, is the method that actually writes the data to the disk.

The `LDocument` class has an implementation for `AskSaveAs()`. Unfortunately, it has not been updated to handle the need for a `::NavCompleteSave()` call under Navigation Services. In order to handle this, we need to rewrite the `AskSaveAs()` command as well as retain a new data member: `myFileDesignator`.

AskSaveAs() in CTextDocument.cp

```
Boolean CTextDocument::AskSaveAs(
  FSSpec& outFSSpec, Boolean inRecordIt)
{

  Boolean    didSave = false;

  StDeleter<LFileDesignator>
    theDesignator(NEW LFileDesignator);
```

```
ValidateSimpleObject_(theDesignator.Get());

theDesignator->SetFileType(GetFileType());
NavDialogOptions*  theOptions =
  theDesignator->GetDialogOptions();
if (theOptions != 0)
{
  theOptions->dialogOptionFlags |= kNavNoTypePopup;
}

Str255 theDefaultName;
if (theDesignator->AskDesignateFile(
  GetDescriptor(theDefaultName)))
{

  theDesignator->GetFileSpec(outFSSpec);
  if (UsesFileSpec(outFSSpec))
  {

    if (inRecordIt)
    {
      SendSelfAE(kAECoreSuite, kAESave, ExecuteAE_No);
    }

    DisposeOf_(myFileDesignator);
    myFileDesignator = theDesignator.Release();

    DoSave();

    didSave = true;

  }
  else
  {

    if (inRecordIt)
    {
      SendAESaveAs(outFSSpec, GetFileType(),
        ExecuteAE_No);
    }

    if (theDesignator->IsReplacing())
    {
      ThrowIfOSErr_(::FSpDelete(&outFSSpec));
    }

    if (myFileDesignator != 0)
    {
      ValidateSimpleObject_(myFileDesignator);
      myFileDesignator->CompleteSave();
      DisposeOf_(myFileDesignator);
    }
```

```
      myFileDesignator = theDesignator.Release();

      DoAESave(outFSSpec, fileType_Default);

      didSave = true;
    }
  }

  return didSave;
}
```

The first thing the code does is to create a new `LFileDesignator` (again, a `StDeleter` is used to prevent leaks). Before interacting with the user, we need to adjust some settings for the dialog. Just as with the `LFileChooser` in `ChooseDocument()` above, the `GetDialogOptions()` will return 0 if it is running under classic dialogs. We adjust the options so that the user will not be presented with a type popup menu. This is consistent with the fact that we only save `TEXT` documents from this application. The code also sets the default name for the file to the name of the window.

The call to `AskDesignateFile()` returns `true` if the user has decided to save the file (and `false` if they cancel the dialog). All the information obtained from the dialog is available through `theDesignator`. The first thing we do is set `outFSSpec` to the `FSSpec` obtained from interaction with the user. Then we determine if the document's existing file is being overwritten or the document is writing to another file. The two possibilities are similar and we discuss them in parallel.

In both cases the first thing the code does is to send the application an Apple Event if the application's Apple Events are being recorded. The actual Apple Event sent is different, but the code is very similar. The parameter `ExecuteAE_No` indicates that the Apple Event should not be executed once it is received. This is appropriate since the command is being handled here and there is no need for the code to save the file twice.

If the user is saving to a different file and that file already exists, then the file needs to be deleted and this is done next (in the second branch). Now it is time to handle the changes to the `myFileDesignator` data member. If we are overwriting our original file on disk, then the old `LFileDesignator` is deleted and the new designator is saved. Since the file is not being closed, the code does not need to call `CompleteSave()`. In the other case (where the file on the disk is changing and the `LFileDesignator` is not 0) the code calls `CompleteSave()` before disposing of the old `LFileDesignator`. Next either `DoSave()` or `DoAESave()` is called. In both of these cases, the function will then return `true` because the file has been saved.

Next we examine the `DoAESave()` method. This method implements a Save As command and calls `DoSave()` to do the actual work of writing to the disk. Almost all of the code is generic and you should be able to use it "out of the box." In fact, only the `OSType_Creator` constant is application specific and could easily be factored into a call to a new virtual function with the name `GetCreatorType()`.

DoAESave() in CTextDocument.cp

```
void CTextDocument::DoAESave(
  FSSpec& inFileSpec, OSType inFileType)
{
```

```
  DisposeOf_(mFile);
  mIsSpecified = false;

  StDeleter<LFile> theFile(NEW LFile(inFileSpec));
  ValidateObject_(theFile.Get());

  OSType theFileType = GetFileType();
  if (inFileType != fileType_Default)
  {
    theFileType = inFileType;
  }

  theFile->CreateNewFile(OSType_Creator, theFileType);
  theFile->OpenDataFork(fsRdWrPerm);
  theFile->OpenResourceFork(fsRdWrPerm);

  mFile = theFile.Release();

  DoSave();

  ValidateObject_(mWindow);
  mWindow->SetDescriptor(inFileSpec.name);

  mIsSpecified = true;
}
```

Since this is a Save As operation, the code first eliminates the existing `LFile` object. This does not delete the file on the disk, but it will close the file if necessary. Notice that this is done even before we know that the new file is valid. My reasoning is that after an attempted Save As command, the goal is to protect the original file's data (and prevent a Save command from clobbering it). The `DisposeOf_` macro will set the `mFile` data member to 0 and setting `mIsSpecified` to false will prompt the user with another `AskSaveAs()` dialog if they try to close the window after a failed Save As command.

Next, the code creates the file on the disk (with another `StDeleter`). Once we know the file has been successfully created and opened, the `CTextDocument` object will take control of the `LFile` pointer (again the call to `Release()` is important to avoid a double-delete). The `DoSave()` command expects that both the resource and data forks of the file are open for writing. Once `DoSave()` writes the data to the disk, the document's window title is updated and its `mIsSpecified` data member is set to true. We now turn our attention to the one method that requires knowledge of the data layout in the files: `DoSave()`.

DoSave() in CTextDocument.cp

```
void CTextDocument::DoSave()
{
  ValidateObject_(myTextView);
  Handle theTextH = myTextView->GetTextHandle();
  ValidateHandle_(theTextH);
  SizetheTextSize = ::GetHandleSize(theTextH);

  StHandleLocker theLock(theTextH);

  ValidateObject_(mFile);
```

```
  mFile->WriteDataFork(*theTextH, theTextSize);

  SetModified(false);
}
```

Saving text files is rather simple: get a handle to the text and write it out to the file's data fork. The last line indicates that the document currently matches the data on the disk. Somewhat anti-climatic, but it works. The actual PowerPlant stationery file saves some information in the resource fork as well, but I will leave that for you to explore on your own.

CONCLUDING REMARKS

Although Opening, Reverting and Saving files requires a significant amount of code, most of the work is done by PowerPlant. In fact, of the methods you need to implement, most of them can be lifted almost verbatim from the PowerPlant stationery files. At this point, I think I have covered the code presented in the PowerPlant Advanced stationery file (with a few omissions that you should be able to work out using the class browser and debugger). In the next article (the fifth of this series), I will spend some time talking about control classes which can be used to liven up your dialogs. Once that topic is covered, I feel I will have covered the essential core of PowerPlant. At this point, I will reiterate my request that people should indicate what topics they would like to see. Here is a short list of topics I have considered: more on menus, more on panes, threads, drag & drop, tables, actions (and undo strategies), Apple Events, contextual menus. I am flexible and willing to look at other topics if they are suggested.

POWERPLANT REFERENCES

I could not find as many references on files as on some of the other topics, there is a single chapter in **The PowerPlant Book** entitled "File I/O". You should also spend some time sifting through the source code of PowerPlant as well as the example files (there is a `StdDialogs` demo as well as a `TextDocument` demo).