POWERPLANT WORKSHOP

By Aaron Montgomery

First Steps

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/`.

MY INTENDED AUDIENCE

This series of articles is intended to be an introduction into the use of the PowerPlant application framework. It is intended to be a gentler introduction than **The PowerPlant Book** supplied on the CodeWarrior CD and more elementary than previous *MacTech* articles (in particular, those by John C. Daub). We will start by building a very basic application in this article and we will add features to it in later articles (upcoming articles are planned on the topics of: debugging, windows, files, dialogs, preferences, and panes).

PowerPlant is big and it would be hard (if not impossible) to try to provide a simple linear introduction to the topic. As a result, these articles are likely to introduce some aspect of the framework without fully explaining how it works. In some cases, the series will make a concerted effort to return to the topic but in other cases the reader will be expected to do a little reading on their own. One consequence of this is that it will probably be worthwhile to re-read earlier articles after finishing with the later articles. Another consequence is that often I will introduce a term without explanation and you may need to read more of the article before it makes sense.

While writing this article, I've assumed that the readers are familiar with the C++ programming language and basic Macintosh API programming topics. In addition to these programming skills, readers are also assumed to be familiar with the CodeWarrior IDE (including the use of the class browser and debugger).

MYSELF

Before you start reading this article, it is best to know where I learned what I know. I program as a hobby (not as a profession) and have been using PowerPlant in my projects for the past three years. Although I consider myself fairly adept with the framework, I don't claim to know all the details. I encourage those who know the framework better to contact me if I make any errors or miss any topics. Probably the best forum for this is the PowerPlant newsgroup and I will also post errata at my web site (see by-line).

WHY LEARN POWERPLANT

PowerPlant is an application framework. You can find a detailed discussion of what this means in **The PowerPlant Book**, but basically, this means that it provides much of the routine code needed for a Macintosh application (in particular, the interface) and allows you to focus on the details of your specific program. At the time that I am writing this, a natural question arises: why bother with PowerPlant when you could use Cocoa? Personally, I decided to continue to use PowerPlant for two reasons. The first of which is my own personal familiarity with C++. Since I only program as a hobby, I don't have the time (or the need) to build that level of familiarity with Objective C and would like to continue to use a language I know. A second reason is that many of the programs I write for educational purposes will be used in environments where OS X is not going to be available for a number of years. PowerPlant allows me to use a single code-base to support a wide variety of platforms (from 68K to G4).

THE TOOLS

This article will assume that you are using CodeWarrior 6 with the net-update to IDE 4.1.0.3 and no other modifications. In particular, it assumes the use of Universal Headers 3.3.2 and no modified PowerPlant files. I believe much of the code I write should also run on CodeWarrior 5 or with newer Universal Headers without much trouble (other than that caused by the Metrowerks' PowerPlant source). If you are trying to use these articles as a guide and run into trouble with a particular configuration, let me know and I will try to suggest a solution.

STATIONARY OPTIONS

The first thing to do with any project is to create a new project file from the Metrowerks supplied stationary. There are four options: Basic, Appearance, Document or Advanced. The Basic option is the simplest and all other options are built on top of it. The Document and Appearance options have identical library source file and include everything in the Basic option as well as the Appearance Classes, the Graphic Utilities, the Grayscale Classes and the Page Controller. The difference between the two versions is that the sample code for the Document option is set up for a basic application supporting the use of documents. The Advanced option contains everything in the Document and Appearance options as well as the Context Menu, the Debugging Classes, the Table Classes and the Threads Classes. This is the option I choose most frequently (the Debugging Classes are so good, they will be the topic of the second article in this series).

For this project, I started with the Advanced option, moved some unused classes out of the regular targets to a side-target (Other Stuff). We'll add them back to the build targets when I start using them in later projects. I have also made some stylistic changes (e.g., requiring explicit use of the PowerPlant namespace and adding a separate file to contain `main()`). Finally, I removed some code so that I will be able to focus on the basics. Much of the removed code will reappear in later articles.

NAMING CONVENTIONS

Before examining any source code, the PowerPlant naming conventions deserve mention. All PowerPlant types defined by `typedef` macros end with a `T`. All PowerPlant classes begin with `L` (for library), `U` (for utility) or `St` (for stack-based). Class methods begin with uppercase letters and use capitalization as a word-break. Class data (static data) begins with the letter `s`

and object data begins with the letter `m`. Calls to the Macintosh API are indicated by a global scope operator (`::`). Local variables are indicated by the prefix `the`. You can find more conventions in **The PowerPlant Book**.

User classes (those that you write) will typically start with a `C` (for class), I also use `Ut` (for utility) and `Au` (for automatic, i.e., stack-based) classes. Class data for my classes begins with `our` and object data begins with `my`. This helps me distinguish between data I am responsible for and data coming from the framework.

BROWSING THE INITIALIZATION CODE

The first code we will discuss is the function `main()` in `main.cp`. One effective way to learn about PowerPlant is to use the class browser to examine a function's definition and commentary. In order to follow along, you will need to make the project with the class browser activated (see the **Build Extras** preference panel for the target). I will be using the PPC Debug target although you may want to use a different target depending on your setup. (I have removed all comments for listing in the magazine.)

main() in main.cp

```
int main()
{
  InitializeHeap(5);
  UQDGlobals::InitializeToolbox();
  ::FlushEvents(everyEvent, nil);
  UEnvironment::InitEnvironment();

  CDocumentApp theApp;
  theApp.Run();

  return 0;
}
```

We'll quickly run through the initialization code in `main()` using the browser. Selecting the `InitializeHeap()` (e.g., by double-clicking the word) and then using **Find Definition** in the **Search** (or by using command-'), we can get the source for the function. PowerPlant commentary is good and the comments before the definition of `InitializeHeap()` describe its purpose: it expands the heap and calls the appropriate version of `MoreMasters()`. The next method is `InitializeToolbox()` and looking at its definition (and code commentary) you can see that it contains the standard Macintosh Toolbox initialization code. Following the toolbox initialization is a `::FlushEvents()` call. This isn't called in the `InitializeToolbox()` method, but commentary in the PowerPlant stationary indicates that failure to call it may lead to problems and so we place it here. The last call in this sequence is `InitEnvironment()`. Looking at its definition (and code commentary) indicates that it will check for things like the existence of the Appearance Manager. Even if you do not use the `UEnvironment` class yourself, PowerPlant uses it and so you need to insure that it is initialized.

We have now reached the class `CDocumentApp`. This class represents the Macintosh application and the object `theApp` represents this instantiation of the Macintosh application. Again, we can use the browser to examine the constructor.

CDocumentApp() in CDocumentApp.cp

```
CDocumentApp::CDocumentApp()
```

```
{
  if (UEnvironment::HasFeature(env_HasAppearance))
  {
    ::RegisterAppearanceClient();
  }

  CTextDocument::RegisterClasses();
}
```

The `HasFeature()` determines if the Appearance Manager is installed. If it is installed, the application registers itself with the operating system. The second statement is a call to the static method `RegisterClasses()` of `CTextDocument`. This is necessary to build the window structure from the resource file and will be discussed in the third article.

STEPPING THROUGH RUN()

The browser is good if the code is very linear, once the code becomes complicated, stepping through the source with the debugger can be a great aid in understanding. In order to do this, add a breakpoint on the line `CDocApplication.Run()` call in `main.cp` and run the application (with the debugger enabled). When the debugger halts at this breakpoint, step into the `Run()` method. I will describe, but won't list, the code from the PowerPlant source. The first thing that occurs is a sequence of initializations for the application: creating a menu bar, setting up the AppleEvent handlers and then calling `Initialize()`. This virtual method should handle initialization routines that need to happen after the menu bar and AppleEvent handlers have been initialized (and hence cannot occur in the constructor). Later we will use this method to create a dynamic menu, for now we fall back on the inherited method. PowerPlant then calls `ForceTargetSwitch()` (described below), initializes the cursor, updates the menu and starts handling events.

EVENT PROCESSING

We have now reached the main event loop. There are two types of events to which the application needs to respond: AppleEvents and menu-commands. The PowerPlant framework handles the dispatching of some basic AppleEvents to virtual methods in the `LApplication` class. The only ones we need to implement here are the methods associated with Open Application and Reopen Application events. PowerPlant dispatches these to the `StartUp()` and `ReopenApp()` methods respectively.

StartUp() and ReopenApp() in CDocumentApp.cp

```
void
CDocumentApp::StartUp()
{
  ObeyCommand(cmd_New, nil);
}


void
CDocumentApp::DoReopenApp()
{
  if ((UDesktop::FetchTopRegular() == nil)
      && (UDesktop::FetchTopModal() == nil))
  {
```

```
    ObeyCommand(cmd_New, nil);
  }
}
```

The basic idea of both methods is the same: pass the responsibility to `ObeyCommand()`. There only difference is that in `ReopenApp()`, the `ObeyCommand()` is only called if no other windows are open. In the actual source file, there is a long commentary about why this is the appropriate behavior.

The method `ObeyCommand()` is a virtual method of the `LCommander` class (from which `LApplication` inherits). Each `LCommander` has a supercommander (well, not quite, the top commander will not have a supercommander). The collection of commanders generated by following supercommanders up to the top commander is called the command chain. At any given time, exactly one `LCommander` object has control of the application, this object is said to be on duty. The call to `ForceTargetSwitch()` that we saw above placed the application object on duty. When `ObeyCommand()` is called, the object on duty should either obey the command or pass it to its base class. A basic example of this in this application is the `cmd_New` when no windows are open. To see how this is handled, place a breakpoint at the top of the `ObeyCommand()` method in `LDocApplication` (you can find this by typing command-' when no text is selected and typing ObeyCommand). Once the breakpoint is set, switch to the application (or start it from the debugger) and select **Open** from the **File** menu. You should step into the `SendAECreateDocument()` and `MakeNewDocument()` methods, the second is supplied by the `CDocumentApp` class.

MakeNewDocument() in CDocumentApp.cp

```
LModelObject*
CDocumentApp::MakeNewDocument()
{
    return new CTextDocument(this);
}
```

That was simple, we just pass all the work off to the `CTextDocument` class. We will leave `CDocumentApp` for a moment and explore the `CTextDocument` class.

CTextDocument() from CTextDocument.cp

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

  myTextView =
    static_cast<LTextEditView*>(
      mWindow->FindPaneByID(kTextView));
  mWindow->SetLatentSub(myTextView);

  NameNewDoc();

  mWindow->Show();
}
```

Much of the code in the constructor handles the user interface and will be discussed in greater detail in the third article. We focus on those calls that concern the command chain. The `CreateWindow()` method creates a `LWindow` from data in the resource fork of the application. `LWindow` is derived from `LCommander` and the second argument in the `CreateWindow()` method sets the window object's supercommander. The window resource contains a `LTextEditView` and `LTextEditView` also inherits from `LCommander` and its supercommander will be set to the `LWindow` being created. The next two lines find the `LTextEditView` and then establish it as being the on duty when the window opens. The remaining two calls are mundane: `NameNewDoc()` uses a very naive method to generate unique window names (which will be improved in the third article) and `Show()` makes the window visible.

Now that we have a non-trivial command chain, we can demonstrate what happens when an object cannot obey a command. In this case, the command works it way up to the `ObeyCommand()` implementation in `LCommander` and you should place a breakpoint in that method. Now make sure a window is open in the application and select the **Close** item from the **File** menu. The first time you stop in `LCommander`'s `ObeyCommand()`, a `LTextEditView` has received the command and is passing it upward to its supercommander. Running the debugger will cause the window to close and the application to continue. This means that the `LWindow` object was able to handle the command.

A good question arises here: If the `LWindow` handled the close event, what happened to the `CTextDocument`? To determine this, place a break point in the `CTextDocument` destructor and close another window. Before looking at the destructor code, examine the stack. By selecting on various locations of the stack you can see that the `LDocument` object's `Close()` method is called which causes the `CTextDocument` object to be destructed.

~CTextDocument() in CTextDocument.cp

```
CTextDocument::~CTextDocument()
{
  try
  {
    TakeOffDuty();
  }
  catch (...)
  {
  }
}
```

The destructor is very simple and only calls `TakeOffDuty()` which is inherited from `LCommander`. This call takes the `CTextDocument` off duty (a good thing since its memory is about to be reclaimed).

To see a more complicated demonstration of the command chain, make sure a breakpoint is still set in `LCommander`'s implementation of `ObeyCommand()`. Now make sure a window is open and chose **Quit** from the **File** menu. This command will need to work up many levels in the command chain. The `LTextEditView` object passes it to the `LWindow` object which passes it to the `LDocument` object which passes it on to the `LApplication` object which causes the application to quit.

The other virtual method that you will need to override when you add (or remove) commands is `FindCommandStatus()`. We will discuss this more fully in a later article when we handle menu management, but for now we only need to worry about the first two parameters. The first, `inCommand`, is of type `CommandT` and is the command whose status is being determined. The second, `outEnabled`, is of type `Boolean&` and will be set to true if the `LCommander` object can handle the command. Each time the menus are displayed, the method `FindCommandStatus()` is called in a manner analogous to `ObeyCommand()`. If any `LCommander` in the command chain can handle the command, the method returns `true` and the menu item is enabled. If no `LCommander` in the command chain can handle the command, the method will return `false` and the menu item will be disabled. Because they are called in identical ways, `ObeyCommand()` and `FindCommandStatus()` need to be consistent in each class.

FindCommandStatus() in CDocumentApp.cp

```
void
CDocumentApp::FindCommandStatus(
    CommandT  inCommand,
    Boolean&  outEnabled,
    Boolean&  outUsesMark,
    UInt16&   outMark,
    Str255    outName)
{
    switch (inCommand)
    {
        case cmd_Open:
        {
            outEnabled = false;
        }
        break;

        default:
        {
            LDocApplication::FindCommandStatus(
                inCommand, outEnabled, outUsesMark,
                outMark, outName);
        }
        break;
```

```
        }
    }
```

The PowerPlant class `LDocApplication` assumes the application will have the ability to open documents. Since we aren't going to implement this until we discuss files (in the fourth article), we need to disable the command in the **File** menu. We simply set `outEnabled` to indicate that the command is never handled and the item will remain disabled. Similarly, the `CTextDocument` overrides its `FindCommandStatus()` method to disable the `cmd_Save`, `cmd_SaveAs` and `cmd_Print` commands. You can trace this method in exactly the same way as we did for `ObeyCommand()`, but that will be left as an exercise for the reader.

CONCLUDING REMARKS

If you've followed along so far, you should be able to start examining some of the command chains in the sample projects that are provided on the CodeWarrior CD. You might also want to start reading through the references I present at the end of this article. The next article will discuss PowerPlant's debugging classes. Following that we will expand on PowerPlant's document classes (windows in the third article and files in the fourth article).

POWERPLANT REFERENCES

The PowerPlant Book on the CodeWarrior CD is an introduction to PowerPlant provided by Metrowerks. Unlike this series of articles, **The PowerPlant Book** focuses on the interface building aspects of PowerPlant first. If you need to obtain a high level of proficiency with PowerPlant, this is the book to work from cover to cover. The **PowerPlant Advanced Topics** book on the CodeWarrior CD continues where **The PowerPlant Book** leaves off. It covers more complicated classes such as the Drag & Drop Classes, Thread Classes and Networking Classes to name a few. The topics are presented so that you will be able to read each chapter as you need it (and not necessarily in a sequential order). Together these books constitute the primary texts for learning PowerPlant.

The PowerPlant Reference on the CodeWarrior CD provides documentation for most of the PowerPlant classes. I found this useful for a short period of time while learning PowerPlant, but after a few months I began to rely more on the class browser, the actual source, and the commentary in the source instead of the reference book.

If you have access to old *MacTech* articles, the "From the Factory Floor" articles in the October, November and December 1998 issues discuss PowerPlant. The first two of these start with a "How would you learn PowerPlant" question. John C. Daub also wrote a series of *MacTech* articles from December 1998 through June 1999 which cover individual components of PowerPlant (but do not cover the basic classes). The *MacTech* issues from 1999 are available online, but the 1998 articles were not up when I checked. You will want to search the article archives at <`http://www.mactech.com/`>.

Finally you might be able to find a PowerPlant class at Metrowerk's CodeWarrior U site <`http://www.codewarrior.com/`>. These courses tend to follow **The PowerPlant Book**.