PowerPlant: A Third-Generation Macintosh Application Framework

by Jim Trudeau
trudeau@metrowerks.com

Abstract

This paper provides a brief overview of some high-level design features in PowerPlant. These design patterns are presented both for their aesthetic beauty, and as an indication of current trends in framework and software design.

Introduction

It is impossible to fully convey the implementation details of an advanced application framework in a single paper. That is what documentation is for. The PowerPlant documentation spans several extensive volumes. I am not going to waste your time trying to duplicate that information here. Rather, my goal is to give you a high-level view of the PowerPlant architecture. When you have finished this chapter you should have a clear understanding of what a good application framework should look like.

Genesis

As Ted Lewis says in the first chapter of *Object-Oriented Application Frameworks*, the Macintosh played a seminal role in the advancement of object-oriented programming [Lewis96]. The object-based nature of a graphical user interface created a strong demand for better solutions to programming problems.

In the beginning, before the widespread acceptance of object-oriented languages, there were application shells in procedural languages for the Macintosh environment. These were the first-generation application frameworks. No such shell is a commercially viable entity today because of the limitations of a procedural approach to object-oriented challenges. For example, procedural code is much harder to modify or adapt to unique circumstances. Object-oriented code can be extended easily by overriding inherited functions. In fact, the desire for easily extensible and maintainable code played a large part in the advancement of object-based languages such as C++.

In response to the need for an extensible application shell, the second-generation application frameworks arrived. Before long, two of these frameworks dominated the market: the world's first commercially successful framework—MacApp—and the THINK Class Library [MacApp, TCL]. However, each inherited a legacy of design constraints forced upon them by the then-dominant procedural languages and the immaturity of object languages. Nevertheless, each injected true object-oriented design into the software development process.

In 1993, a third-generation framework arrived on the scene—PowerPlant. Gregory Dow, PowerPlant's chief architect and designer, was also the spark behind the original TCL. Fully aware of the compromises required at the time of TCL's initial development, and with years of practical framework design experience, he wanted to do it better.

Applying everything he had learned, he took the concept of an application framework through another iteration. PowerPlant has no baggage from earlier versions originally implemented in languages lacking object-oriented characteristics. PowerPlant is a pure C++ framework that takes full advantage of all the object-based features of the language [Metro96]. You will see how as we examine the design principles that guided the development of PowerPlant.

As a result, although PowerPlant is the youngest of the "big three" in Macintosh application frameworks, PowerPlant is arguably the most flexible and easiest to use. These advantages grow directly out of a fundamental design decision made early in the process of developing PowerPlant. Dow wanted to create a highly-modular collection of subsystems that could be used independently of each other, thus breaking the monolithic pattern of second-generation frameworks. However, he also wanted the modules to form a seamless and complete application framework. How he accomplished these goals becomes clear as we discuss PowerPlant's design features.

Design Features

The PowerPlant application framework demonstrates the judicious use of the following fundamental features of object-oriented design:

- Multiple inheritance—a mix-in architecture with multiple base classes.
- Factored design—classes are as independent as possible.
- Factored classes—classes are as small as possible.
- Factored behavior—individual behaviors within a class are carefully separated into simple, component parts.

These principles taken together ensure that the PowerPlant application framework is modular so that you can learn it quickly, yet powerful enough that you can create full-featured, world-class applications. We should examine each of these principles to see how they affect the PowerPlant framework, and why they make PowerPlant easier to learn and use.

Mix-in classes

PowerPlant takes full advantage of C++'s support for multiple inheritance. As a result, the class hierarchy in PowerPlant is a series of small, loosely-connected trees rather than a monolithic monster tree.

PowerPlant has many classes whose sole purpose is to be mixed into other classes by multiple inheritance. (PowerPlant naming conventions use L for major classes, and U for utility classes.) Such a mix-in class encapsulates state and behavior that are common to an otherwise disparate group of classes. For example, you might want a wide variety of different classes to send messages. In PowerPlant, message-sending behavior is isolated into the LBroadcaster mix-in class. Any class that requires this behavior multiply inherits from LBroadcaster. In this way you can add or "mix-in" the desired behavior to any class.

Among the common PowerPlant mix-in classes are:

- LCommander—for objects that respond to commands.
- LBroadcaster and LListener—for objects that broadcast or receive messages.
- LPeriodical—for objects that receive time on a regular basis.
- LAttachable—for objects to which attachments can be connected.
- LModelObject—for objects that are scriptable via Apple events.

We will visit each of these classes (except LModelObject) in some detail later on.

To see the direct advantage of multiple inheritance in object design, let us look at two PowerPlant classes related to text display: LEditField and LCaption. Both of these classes represent visual objects, so they inherit primarily from the base PowerPlant visual class,

LPane. An object of either class displays text. An LEditField object allows the user to interact with the text by typing, copying, pasting, and so forth. LEditField also supports a flashing text-insertion cursor. LCaption, on the other hand, simply displays a static line of text and allows no user interaction.

As shown in Figure 1, LEditField inherits the command-related behavior it needs from LCommander. It inherits the behavior it needs to flash the cursor from LPeriodical. Both LCommander and LPeriodical are PowerPlant mix-in classes. As mentioned just a moment ago, a mix-in class encapsulates particular behavior (in this case command behavior and periodical behavior) that is added to disparate classes by multiple inheritance. LEditField needs this behavior, so in the PowerPlant architecture the behavior is mixed into LEditField.

By contrast, LCaption has no need for either kind of behavior, so it does not inherit from either mix-in class. This keeps LCaption as lean and as simple as possible.

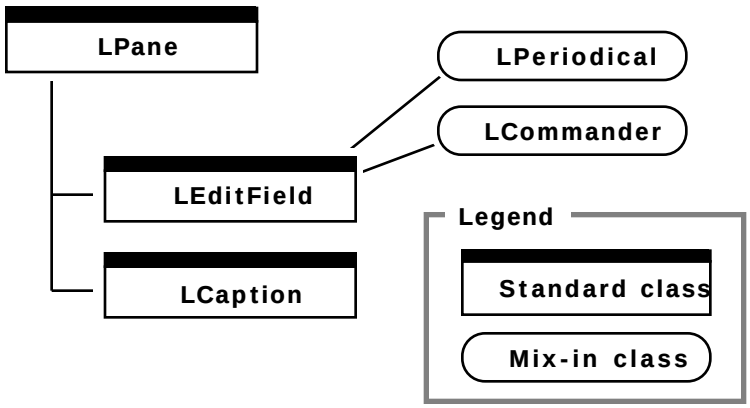

Figure 1. LEditField and LCaption inheritance hierarchies

In a single-inheritance hierarchy, commonly-used behaviors must appear high in the chain so that those classes that need the behavior can inherit them. Features tend to be piled onto classes not because all classes need them, but because some classes need them somewhere down the chain. This results in bloated objects filled with data members and member functions you never use. In a single inheritance framework, the caption class inherits all sorts of useless behavior, as shown in Figure 2.

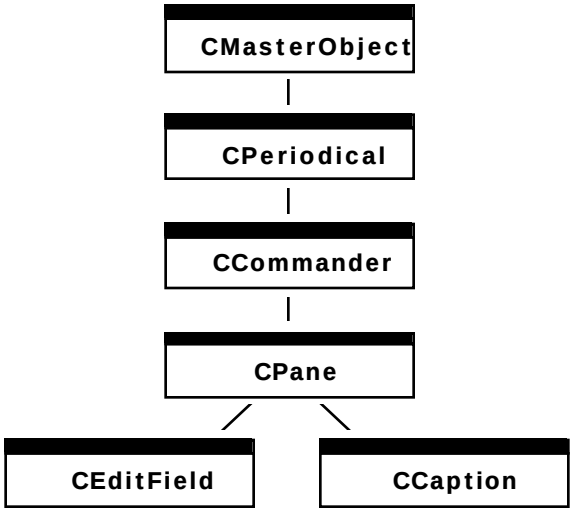

Figure 2. A hypothetical single-inheritance hierarchy

Factored Design

PowerPlant classes are based on the principle that isolating classes from one another reduces complexity and enhances code reusability. The mix-in architecture is the principal reason why PowerPlant can implement its second design goal—to keep classes as independent as possible.

Beyond the important classes cited in the previous section, PowerPlant has a substantial number of small base classes that you can use in a whole variety of circumstances—without using any other part of PowerPlant. In some cases, you do not even need a Macintosh! Some elements of PowerPlant are platform-independent, containing pure C++ code that makes no reference to the Mac OS. This design excellence makes PowerPlant a treasure trove of reusable code.

For example, implementing menus and a menu bar is a common task for a Macintosh application. The PowerPlant LMenu and LMenuBar classes refer to each other, but to no other class. If you want to take advantage of PowerPlant's menu creation services, you can use those classes in your own projects without using any other part of PowerPlant.

If you need to maintain dynamic lists in your project, you can use the LArray and LArrayIterator classes independently. These two classes are a practical realization of the iterator design pattern [Gamma95]. As you will see in the coming discussion, PowerPlant uses this pattern internally in many cases where an object includes an aggregation of sub-objects. However, you can remove the pattern from PowerPlant and use it effectively in any C++ code.

There are many more examples. If you want to filter keystrokes before processing them, the UKeyFilters class has a variety of filters ready for your use. The UDebugging class encapsulates powerful debugging features. Each of these classes is independent of the rest of PowerPlant.

The list goes on and on. PowerPlant consists of approximately 175 classes, containing 1,500 functions (including all the overridden functions in subclasses) implemented in about 25,000 lines of source code. Many of the classes are small, utility classes whose purpose is to make common programming chores simple. There are, quite literally, dozens of useful classes that have been purposely designed to be completely or almost completely independent of the rest of PowerPlant. In many cases, these classes implement a recognized design pattern. We are going to examine some of these patterns a little later.

The principal base classes in PowerPlant each form the trunk of a separate class hierarchy. They are:

- LPane—classes representing visual items in the interface, including windows, views, and controls.
- LMenu and LMenuBar—support for the Mac OS menu architecture.
- LAction—classes that encapsulate commands, used for do, undo, and redo behavior.
- LAttachment—classes representing objects that are attached to other objects to modify behavior at runtime.
- LArray and LArrayIterator—PowerPlant's list mechanism.
- LStream—support for data stream in all contexts.
- LFile—support for the Mac OS file system.
- LDocument—connects stored data (files) with windows (views on data).

A little earlier we mentioned several mix-in classes in PowerPlant. Various classes within the main hierarchies multiply inherit from the mix-in classes when necessary. The result is that a complete map of the PowerPlant class inheritance is not a monolithic tree. It is instead a web of interacting modules.

Of course, in an application framework some dependencies are unavoidable. For example, an LDocument object expects to find and use LFile and LView objects. However, by and large these hierarchies and the mix-in classes are completely independent. Each of these independent classes meshes seamlessly as a vital component part of a complete application framework. The whole is decidedly greater than the sum of its parts. However, the parts remain fully and completely accessible to you as independent, reusable code modules.

This design encourages what chief-architect Dow calls a buffet-style approach to framework use. "You look over all the classes, pick the ones that look interesting, and then go back for more later. Furthermore, the wide variety of classes tempts you to experiment with features that you might not have tried on your own [Metro96]."

From a vantage point high above the PowerPlant design you can see the clear difference between a monolithic application framework that requires you to buy the entire store every time you want to use one tool, and a well-designed application framework based on multiple inheritance and a factored design where you can simply take the tools you need.

Factored Classes

The third principle that guides the PowerPlant designers is that behavior should be placed as low as possible in the class hierarchy.

This process goes hand-in-hand with the factored design of PowerPlant we discussed in the previous section. The distinction between a factored design and factored classes is simple.

A factored design looks at the problem domain in broad perspective. The designer determines what parts of the problem can be separated from other parts. The designer then creates modular class hierarchies for each separate part of the problem domain.

The process of factoring class behaviors, on the other hand, is more precise. The designer must determine where a particular behavior should appear within a particular class hierarchy.

A careful analysis of the design of a typical Macintosh application identifies the behaviors that various objects in the application must provide. The designer determines what a window does, what a scroll bar does, and what a radio button does, for example.

By analyzing the behavioral demands on the various identified objects in the system, the designers of PowerPlant have carefully factored behavior so that it appears only when necessary. General behavior appears in base classes, so it can be inherited by all those subclasses that need it. Behavior that is more specific to a certain kind of object does not appear in a class declaration until necessary.

This does not mean that some classes are not large and complex. The class that defines window behavior for example, LWindow, has 60 member functions declared in the class, and inherits about 200 more from three base classes.

However, even in a complex class like LWindow you can quickly recognize that certain member functions come from the various base classes from which LWindow inherits, as shown in Figure 3. This logical structure allows you to look at the most complex class as an aggregation of its ancestors, making the whole that much more understandable.

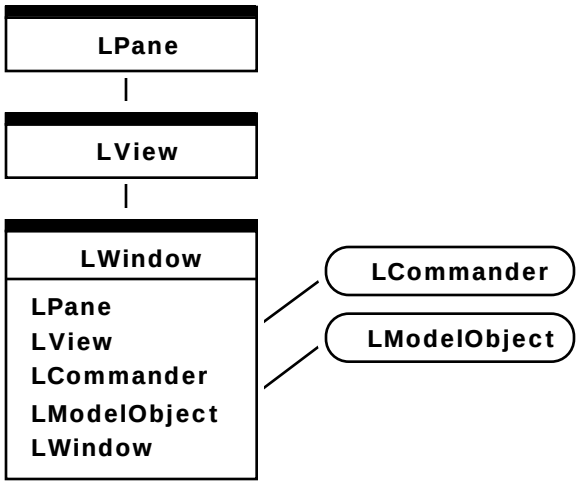

Figure 3. LWindow is the sum of its ancestors, plus its own behavior

Factored Behavior

The fourth principle that guides the creators of PowerPlant is that complex behaviors should be factored into simple, constituent parts. Once again, like the emphasis on factored classes and a factored design, factoring behaviors into their component parts continues the trend toward small, simple building blocks that you see throughout PowerPlant.

To implement this principle, member functions that affect an object are usually split into two parts. We will call them the setup part and the action part. The setup part handles any state-testing or adjusting that must happen before the action takes place. The action part implements the desired behavior.

An example will help here. A pane in its most general sense is a rectangular area in which some drawing occurs. LPane is the base class for all classes that describe visual objects in PowerPlant. LPane declares two member functions, Draw() and DrawSelf(). Together these functions draw the contents of the pane at runtime, whenever the object needs to be updated visually. The Draw() function is the setup routine. It makes sure that the pane is visible and that its coordinate system is set up properly. Then it calls DrawSelf()—the action routine that does the actual drawing. Each subclass of LPane inherits both functions.

Very few subclasses of LPane override the Draw() function. Setting up the drawing environment is a task that is identical for most pane classes. However, all classes of LPane override DrawSelf(). The DrawSelf() function does the actual drawing. Because the appearance of each pane varies, each requires its own version of DrawSelf().

Because the PowerPlant designers factored drawing behavior into separate functions—setup and action—you do not have to write housekeeping code every time you override drawing in a derived pane class. PowerPlant does everything it can to maximize the reusability of code. Figure 4 illustrates this principle.

You will find this type of factoring throughout PowerPlant for all kinds of very common behaviors including drawing, responding to a mouse click, activating an object, executing a command, and so forth.

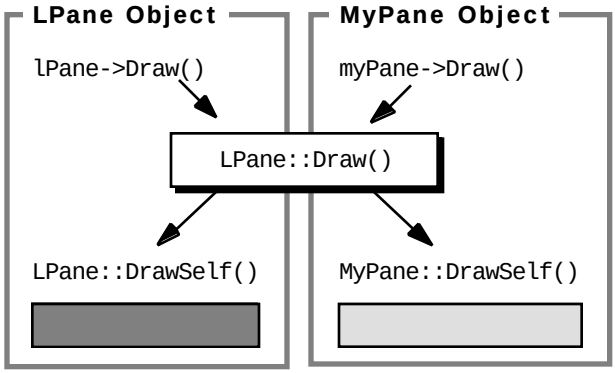

Figure 4. Reuse of code because of factored behavior

Learning How to Use a Framework

No matter how powerful a framework is, it does not help you very much if you cannot figure out how to use it. Learning a framework is not an academic exercise. You want to master the framework so you can increase your productivity and your ability to write better, more reliable software.

Unfortunately, there is a hurdle between you and your goal. It is safe to say that learning how to use a framework is much more difficult than actually using it. Whatever framework you choose, you must learn the individual peculiarities of that framework. What programming idioms do the designers use? How does the framework create a window? What are the names of the commonly-used functions? What are their parameters? Which functions are commonly overridden? When a framework has hundreds of classes and thousands of functions, these are not trivial questions.

As a third-generation framework, PowerPlant makes this task a lot easier. One of the greatest potential benefits a programmer will derive from PowerPlant's extraordinary design is that PowerPlant is inherently easier to learn than other frameworks. Why? Because you can learn it incrementally.

A typical second-generation framework is a monolith. You must swallow it almost whole if you are to use it at all. PowerPlant really is a buffet. You can pick and choose the part of PowerPlant that interests you most, and learn it without regard to the other parts of the framework. This approach minimizes the amount of material you must master at any one time.

PowerPlant's modular design makes the framework more accessible to you as a busy programmer. If it is more accessible, easier to learn, and more flexible, then it is more likely that you are going to master the framework and gain the advantage of reusable code.

Code reuse is one of the most important goals of object programming [Goldstein92]. Frameworks are the most powerful and visible means of achieving that goal [Pree95]. Each of PowerPlant's four design principles—a mix-in architecture, factored design, factored classes, and factored behavior—is aimed at maximizing the reusability of PowerPlant code. PowerPlant is not just a blank Macintosh application. PowerPlant is a whole suite of tools that you can use in a wide variety of programming projects. Let's see how these design principles bear fruit in the practical world of code.

Design Patterns

We are going to look at four programming challenges and how PowerPlant's design implements an elegant solution for each one. We will look at the actual classes involved and how they work together. We will also look at the degree of interdependence between those classes and the rest of PowerPlant. The programming tasks we examine in this section are:

- Handling commands—the command hierarchy in PowerPlant.
- Inter-object messaging—communication between any two disconnected objects.
- Periodical behavior—implementing idle time and other repeated processes.
- Attachments—runtime modification of objects by altering composition.

Handling Commands

PowerPlant provides complete and fully-realized event retrieval and identification, as you would expect from any application framework. Things get interesting after the event is retrieved and identified.

In PowerPlant, most events generated by the Mac OS fall into two principal categories: commands and clicks. A command is generated by a keystroke or the user choosing a menu item—whether by typing a key equivalent or choosing a menu item with the mouse. A click, on the other hand, occurs when the user clicks the mouse somewhere in the content area of a PowerPlant pane.

This separation of commands and clicks means that PowerPlant has two different event handling hierarchies—one for commands, and one for clicks. We are going to discuss only the command handling mechanism.

A command, then, is really a message from the user to the application. The application—or more precisely, some object in the application—must respond to the message. PowerPlant's command handling mechanism is an excellent, practical example of the chain of responsibility design pattern [Gamma95].

In PowerPlant terminology, an object that can respond to commands is called a commander. In the chain of command, those higher up are called supercommanders and those lower down are called subcommanders. Any given commander may be both a supercommander to its own subcommanders, and a subcommander of some other supercommander. No object may have more than one supercommander, but it may have several subcommanders. Therefore, the command hierarchy is a tree, as shown in Figure 5.

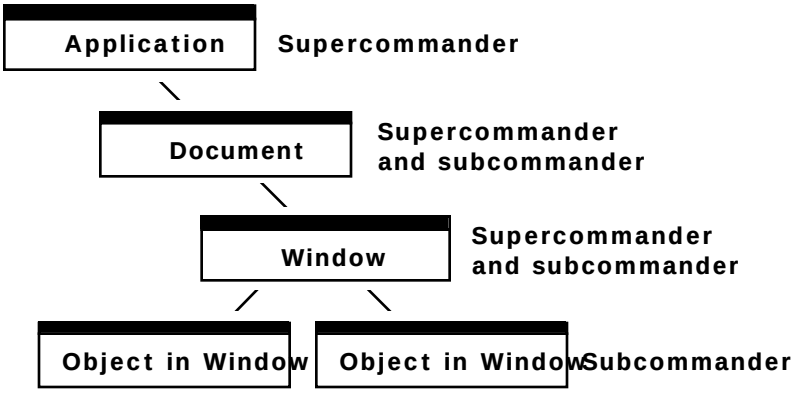

Figure 5. PowerPlant command hierarchy

A commander may have no subcommander at all if that particular commander is a leaf on the command tree. And one commander—the top commander—has no supercommander. In PowerPlant the top commander is the application object.

PowerPlant dispatches commands from the bottom up. The application keeps track of a target object. The target object is the currently active command object destined to be the recipient of all commands. When a command is dispatched, it goes directly to the target object, which is always a commander.

If the target is incapable of handling the command, it passes the event up the command chain to its supercommander. The supercommander may handle the event, or pass it on to the next commander up the chain. Ultimately, if no object handles the event the application object receives the event back, and must handle the event itself. Figure 6 illustrates this process.

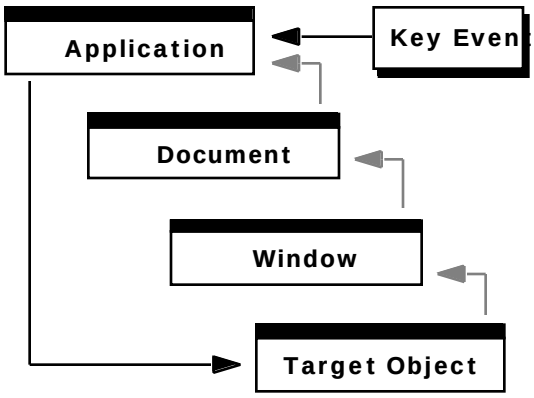

Figure 6. Chain of responsibility in PowerPlant

At any moment, one and only one commander is the target commander. The target may be anywhere vertically in the command hierarchy, it need not be a leaf commander. The target can be any commander object at any level in the command hierarchy.

The principal advantage of the bottom-up approach to the chain of responsibility is that the target object can adjust the context of the application to match the object's own capabilities. Because it gets events first, the target can set up menus properly to reflect its behavior. If the target is an editable text object for example, it might enable a font menu.

This gives you tremendous flexibility. You can put a great deal of power down deep into the leaves of your command hierarchy, and let the commanders take care of things for you. This makes the central control system a lot simpler. If you add a new kind of object, you do not have to redesign the control system, you simply give that new object the knowledge necessary to adapt the entire application to its needs.

As an additional advantage, tracing the command chain is a lot simpler from the bottom up. Each supercommander may have several subcommanders. Figuring out which way to go from the top down is a non-trivial task. Figuring out which way to go from the bottom up is simple. There is only one path because each commander has one supercommander.

Finally, a bottom-up approach is usually the most efficient method of handling a command. In most cases, the target object is the object with which the user is dealing, and in most cases it is capable of handling the command. For example, if the target is an editable text object, it is likely that the user is typing. Every keystroke goes directly to the object responsible for processing the key. This makes the application more efficient and responsive because the most likely destination for the command—the target—has the first opportunity to respond.

This design works well no matter what the source of the command. The command might come directly from the user at the keyboard, or arrive via a network connection. The design also works well regardless of the nature of the command. If the command operates on data (for example, a paste operation), the target gets first crack at the data. If the command is inappropriate for the target (for example, an attempt to paste data the target does not understand), the target can choose to pass the command to its supercommander for processing.

LCommander is the base class from which all commander objects inherit. It has functions for command chain maintenance—changing supercommanders, or adding or removing subcommanders. Each commander can also manage the target object. A commander has member functions to set the target, be the target, not be the target, and so forth.

A commander may be on or off duty, and has functions to manage the duty state. This is an important concept, because an off-duty commander will not receive or respond to commands. When off duty, however, it is important to keep track of which subcommander (if any) was the target the last time this particular commander was on duty. Then, when this commander resumes duty, the framework can activate the correct subcommander as target. This is called the latent subcommander. Each commander has the ability to set or change its latent subcommander.

LCommander is a PowerPlant mix-in class. There are about a dozen classes that inherit from LCommander in three separate PowerPlant class hierarchies: the application class hierarchy, the document class hierarchy, and several scattered subclasses of LPane.

LCommander itself requires very little of PowerPlant, and is in theory an independent module. It requires the services of PowerPlant's array and attachment classes, and no others. As a practical matter, however, if you are going to use PowerPlant's command architecture, you are likely to be using the visual architecture as well, and thus working with the largest and most complex part of PowerPlant.

The remaining design patterns we discuss are quite different in scope. Each provides a fundamental service, and each is an independent module within PowerPlant. Each is a useful weapon in your programming arsenal.

Inter-Object Messaging

As you know, there are many situations where you want one object to know that something happened to another object. In other words, one object is dependent on the state of another object. For example, the user clicks a check box to activate some feature. As a result, new objects become enabled and other objects become disabled. This kind of situation arises all the time.

How do you tell one object that something happened to another, unrelated object? There are all sorts of potential solutions to this challenge. You can create complex messaging hierarchies, you can make classes friends of each other and send direct messages. One solution to this problem is known as the notifier, where one object notifies other objects that something has happened. This design pattern is also known as the observer [Gamma95]. PowerPlant uses a simple and elegant version of the observer pattern that is virtually unlimited in scope yet extraordinarily easy to implement.

The two PowerPlant classes involved are LBroadcaster and LListener. These classes correspond to the subject and observer in the observer design pattern, and to the notifier and responder in other frameworks.

LBroadcaster is another extremely simple mix-in class. It has three significant functions: AddListener(), RemoveListener(), and BroadcastMessage(). LListener is equally simple. It has functions to start and stop listening (so you can turn off listening temporarily if you wish), and a function named ListenToMessage().

An object that inherits from LBroadcaster has the ability to broadcast a message. Each broadcaster has an array of listeners—objects that inherit from LListener. Whenever it

broadcasts a message, the broadcaster uses an array iterator to call each listener's ListenToMessage() function. A practical example shows how useful object messaging can be.

PowerPlant has a group of "control" classes. These classes describe visual interface elements that respond to clicks and cause actions to occur. For example, various kinds of buttons, a scroll bar, and a check box are all control objects.

PowerPlant control objects are broadcasters. When the user clicks a control, it broadcasts a message to its listeners. That message may contain arbitrary information. The message might identify the broadcaster, pass necessary state information, and so forth.

Objects that are dependent upon the state of the control must be LListener objects. Inheriting from LListener gives an object the ability to receive a message from a broadcaster. Each listener registers with the control object on which it depends. When a message is sent, the listener receives the message and may respond. Figure 7 illustrates the relationship between broadcaster and listener.

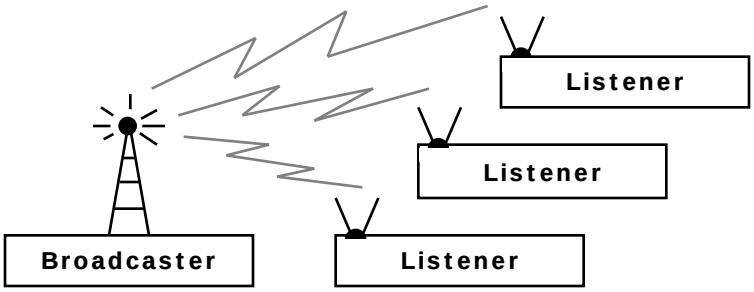

Figure 7. The broadcaster/listener mechanism

The design of the messaging system is remarkably simple, remarkably powerful, and very loosely coupled. The broadcaster has an array of listeners, but it does not know the nature of those listeners. The listener might be a window, a commander, a pane, or any other object that also inherits from LListener. The true nature of the listener is irrelevant to the broadcaster.

The array of listeners is dynamic, and can change at any time with no effect on the broadcaster. A listener may receive messages from any number of broadcasters, but does not need to know anything about the internal workings of any broadcaster. It receives all the information it needs in the message.

This entire mechanism, except for a dependence on the array and array iterator, is a completely independent PowerPlant module. You can bodily remove the messaging system from PowerPlant and drop it into any project to take advantage of this powerful design pattern.

Periodical Behavior

As a programmer, you know that certain types of objects need attention repeatedly. A classic example is the editable text object that flashes a cursor. Regardless of what the rest of the application is doing, this object needs periodic attention so it can manage its responsibilities.

Some periodical objects perform non-critical tasks. You might want to give such an object time only when the rest of the application is idle and unoccupied. Other tasks require regular and repeated attention.

PowerPlant allows you to treat a periodical object either way. Like messaging, the PowerPlant mechanism for implementing periodical behavior is remarkably simple, yet extraordinarily powerful.

Any object that performs a task that requires periodical attention inherits from LPeriodical, another PowerPlant mix-in class. This class maintains two lists of periodical objects. Those that receive time when the application is idle are called idlers. Those that receive time after every pass through the event loop are called repeaters.

Every periodical object has functions to start and stop idling, and to stop and start repeating. The same object may be in either, neither, or both lists at any moment, the choice is yours. Each object also has a SpendTime() function. This is the only function you must write. In this function you implement whatever behavior you want to happen when the object receives time from PowerPlant—either while idling or repeating. The same function serves for both purposes.

The SpendTime() function can do just about anything you want. You can maintain a progress bar, run a simple animation, blink a cursor, and so on. PowerPlant's internal use of the periodical mechanism demonstrates this flexibility.

There are four classes in PowerPlant that inherit from LPeriodical. LTextEdit and LEditField inherit from LPeriodical so that they can blink the text cursor. LMovieController is a periodical so that it can manage QuickTime movie events properly. LGrowZone uses its SpendTime() function to check on the application's memory reserve. It also warns the user of memory problems if necessary.

PowerPlant uses the same mechanism to implement cursor updating, special event processing, and memory management. Your use of LPeriodical is limited only by your imagination. If you have a situation where an object needs time repeatedly, make it a periodical. Design the SpendTime() function to perform the necessary tasks. Install the object in the appropriate queue—either repeater or idler—at the appropriate times, and remove it from the queue when finished.

The periodical object is not required to respond every time SpendTime() is called. Your object can keep track of the passage of time, and only do something if a required interval has passed.

The classic example is a clock. Assume your application displays a timer or clock of some sort that you update every minute. Your timer object might get called several thousand times during that minute, but only act when a full minute has passed.

Once again, this mechanism is (except for requiring that extraordinarily useful array iterator mechanism) a completely independent module in PowerPlant. The PowerPlant event loop calls the necessary functions so that periodicals receive time at the appropriate moments. If you want to use this same mechanism in a non-PowerPlant context, you would have to replicate that functionality in your own event loop. Other than that, this remarkable mechanism is yours free and clear, without any other framework overhead.

Attachments

The final design pattern we are going to discuss is PowerPlant's attachment mechanism. This mechanism implements a design pattern seen in MacApp [Wilson90] and widely known as the decorator pattern [Gamma95]. In MacApp, decorator objects are called adorners. In PowerPlant they are called attachments.

The classic use of the decorator design pattern is to attach drawing behavior to a visual element to modify its appearance at runtime. For example, when the user selects an object in a drawing program, the application may display four tiny squares at the corners of the object. An attachment to the object can handle this drawing task.

From a high-level perspective, however, what an attachment does is modify the object by modifying its composition. You connect or disconnect attachments from a host object whenever necessary, thus modifying the runtime behavior of the object. Drawing is simply a form of behavior. Other kinds of behavior can be modified by an attachment as well.

This is where PowerPlant takes the decorator design pattern to new heights. It takes the concept of modifying behavior far beyond decorating a visual object. It should come as no

surprise to you that the attachment mechanism in PowerPlant is very general—and very effective.

There are two parts to the attachment mechanism in PowerPlant: the objects to which you connect the attachments (the hosts), and the attachments themselves. There are two corresponding base classes, LAttachable and LAttachment. We examine LAttachable first.

Figure 8 illustrates the PowerPlant classes that inherit from LAttachable. Every visual, command, and event-related object in PowerPlant can be a host and have attachments.

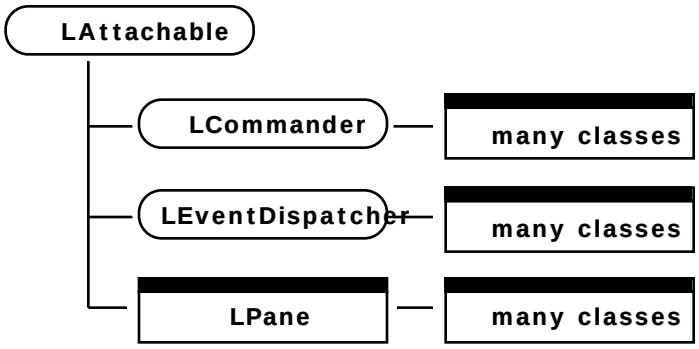

Figure 8. The LAttachable class hierarchy

Each host maintains an array of its own attachments. (There's that list mechanism again!) The host—an LAttachable object—can add or remove attachments from the list, allowing you to modify the list of attachments at runtime.

At certain well-defined moments, PowerPlant tells the host object to walk through the list of attachments. PowerPlant sends a specific message that identifies the task that the host is about to undertake, and sends any data that might be required to fulfill the task. The task might be to update menu items, draw, print, respond to a keystroke, or any number of other possibilities. The task message is sent before the host performs its principal, underlying task.

In response to the message, the host tells each attachment in its list to do whatever it is the attachment does. The relationship between host and attachment is very loose. The host has no knowledge of the nature of its attachments or their purpose.

The attachment object determines if the message it receives from the host is one to which it should respond. If the attachment is designed to respond to the message, the attachment executes. The attachment also returns a Boolean value that determines whether the host object should execute the original task, or whether the attachment has fully handled the task.

To implement an attachment object's functionality you write one function—ExecuteSelf(). The attachment needs no knowledge of the nature of the object to which it is attached. It receives any necessary information in the message, and acts accordingly. The attachment can decide whether to respond to a given message, and how it should respond.

PowerPlant comes with several ready-made attachment classes for handling keyboard navigation, framing or erasing a pane, updating menu items, and so forth. You can create your own attachments easily.

In general, an attachment is an excellent solution when you have an independent behavior that you wish to implement for a variety of panes or commanders, either in the same project or in different programming projects. Putting this behavior in an attachment makes your code eminently reusable.

An attachment is also an excellent solution when you want to modify the behavior of a pane or commander dynamically. You can add and remove attachments at will depending upon the

application's context. If you think of an object as the sum of itself and its attachments, you can modify the composition of the object dynamically by adding or removing attachments.

When considering the utility of attachments, consider the three principal kinds of objects that can host attachments: event handlers (the application object), commanders (command handlers, including the application object), and panes (visual objects).

Event-handling is the most powerful use for an attachment. After an event is retrieved but before it is dispatched, the application's attachments get a crack at it. You can do anything you want with the event. After you handle the event, you can short circuit event dispatch or allow the event to be handled normally. For example, you can add a menu to an application by the simple expedient of connecting an attachment to the application.

A commander's attachments get first crack at all command messages. You may design an attachment to handle a specific kind of command. Rather than write the code directly in a command class, if you decide a commander should respond to a particular kind of command you simply hook up an attachment that does the work. For example, you could use a command-level attachment to create a demonstration version of an application. The attachment intercepts certain commands and disables them.

With respect to panes, attachments have an opportunity to execute before drawing, clicking, and cursor adjustment. You may do some complex drawing in a pane. If you want any unique behavior to occur when a pane is clicked, create an attachment to implement the behavior. The possibilities are endless.

Like many of the other design patterns, the attachment mechanism is an independent PowerPlant module that can be removed from the rest of PowerPlant and used in another context.

PowerPlant's use of attachments reflects an extraordinarily simple, robust, and unbounded design pattern. This kind of elegance can be found elsewhere in PowerPlant—for example, in the broadcaster/listener messaging mechanism. But nowhere else are true strength and simplicity so well combined.

The four design patterns discussed here—chain of responsibility, messaging, the periodical, and the attachment—are a fair sampling of the kinds of design patterns you find in PowerPlant, but there are many more: some independent, some integral to the framework.

Creating a Visual Interface

This overview of PowerPlant would not be complete without a brief inspection of Constructor—PowerPlant's visual interface builder. Constructor allows you to choose visual elements (panes) from a palette of possibilities. You interactively add or remove objects to a view such as a window. You can size and position panes in the window, and set various characteristics reflecting the state and/or contents of the pane. You can easily create custom pane objects with custom data sets as well. Custom objects are easily transportable from one project to another.

Using Constructor as the interface builder, you design the appearance, initial state, and relationships of each of the visual elements in your application. In addition, you specify the class of each of these visual objects. Constructor stores all this information (including class identity) in a special resource type known as a "PPob," (pronounced pea-pob) which stands for PowerPlant object. In a very real sense, Constructor is a visual editor for a PPob resource.

The data in the PPob resource fully describes the view and all of its contents. This data includes the hierarchical relationships of panes contained within other panes, and each pane's initial state.

Constructor does not generate code per se, it generates a resource. You must write one line of code to tell PowerPlant to build the view based on the PPob resource created in Constructor. At runtime, this single function call tells PowerPlant to read the PPob resource stream. In response, PowerPlant's default code parses the data in the stream. As it reads the stream, it encounters

data for each object in the view. The first datum for an object identifies the class of object about to be created. In response, PowerPlant calls that class's constructor function.

The constructor function reads from the resource stream and initializes the object based on the data in the PPob resource. As a result, each object's state is set to the initial values you decided upon in Constructor. In addition, each object has the correct hierarchical relationship with superviews and subviews. Each object is the correct class, complete with all the behaviors expected of an object of that class.

In response to your single line of code, PowerPlant creates not only the overall view, but every single object in the view hierarchy—including custom objects. Because this mechanism is built right into the default behavior of PowerPlant, there is no need for Constructor to generate extensive code to create the visual interface.

Because one of the primary purposes of an application framework is to create and manage the visual interface, the Constructor/PPob resource is a vital component of the PowerPlant approach to programming.

Figure 9 shows Constructor's project window. As you can see, in this particular example there are four separate PPob resources for different windows in this application. Constructor can also describe other kinds of resources, including Mac OS menus, resources to describe text (font, size, style, alignment, and color), and resources for custom visual objects., as well as other resource types.

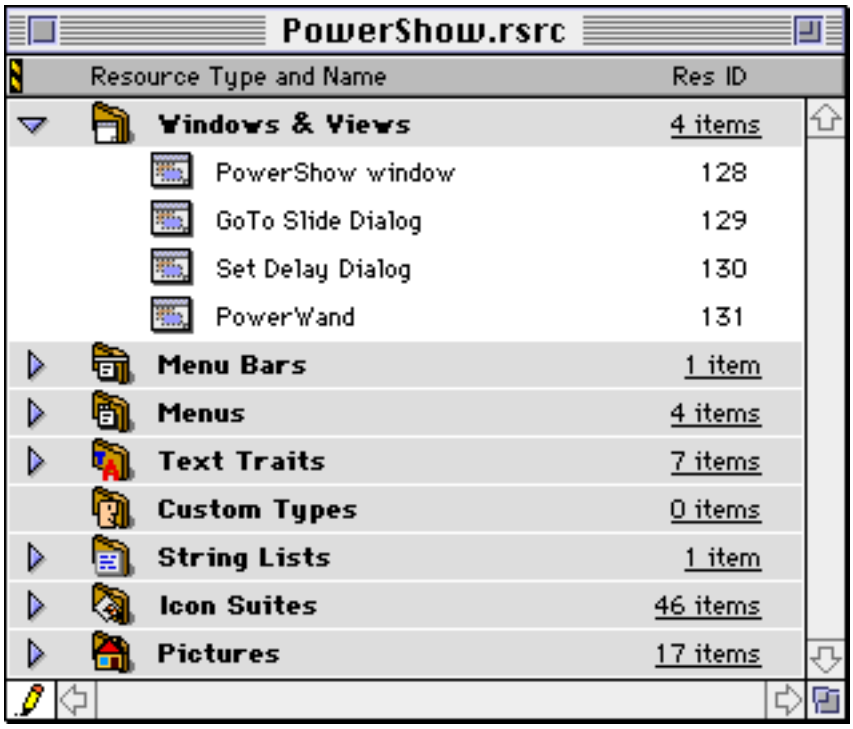

Figure 9. The Constructor project window

The Future

Although we have covered some of the high spots on this tour of PowerPlant, do not lose sight of the fact that all the landscape in between is covered with real, robust, C++ code.

While some of the modules within PowerPlant are platform independent, PowerPlant as a whole is a Macintosh application framework. It is not intended for cross-platform

development. However, all of the features you expect to find in a Macintosh application framework are there, including several features we have not discussed such as:

- multi-threaded processes
- managing actions with undo and redo
- scriptability with Apple events
- support for interface features like drag and drop
- network-related classes providing one wrapper for both MacTCP and OpenTransport
- memory management and debugging

Solutions and support are present in PowerPlant today for each of these programming challenges.

In addition, PowerPlant is not static. The C++ language and C++ compilers are still evolving. Old solutions to programming challenges are replaced when better solutions are developed. New features are added as the Mac OS improves. For example, PowerPlant will continue to support new features as they are introduced into the Mac OS, such as the Appearance Manager in System 8.

In this brief examination of the PowerPlant design you have seen how this framework is a cohesive collection of various design patterns, each independently realized in elegant C++ code. These patterns overlap, intermingle, and work together synergistically to create an application framework that is powerful, flexible, and relatively easy to master.

In the closing comments of their fine work, *Design Patterns*, Erich Gamma et al. looked to the future and said, “The best designs will use many design patterns that dovetail and intertwine to produce a greater whole [Gamma95].” The promise of the future is here today.

Bibliography

[Gamma95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley, 1995.

[Goldstein92] N. Goldstein, J. Alger, *Developing Object-Oriented Software for the Macintosh*. Addison-Wesley, 1992.

[Lewis96] T. Lewis, *Object-Oriented Application Frameworks*. Manning Publications, 1996.

[MacApp] MacApp is available from APDA, Apple Computer Inc., P. O. Box 319, Buffalo, NY 14207-0319.

[Metro96] J. Trudeau, *The PowerPlant Book*. Metrowerks Corp., 1996-1997 (available as part of the PowerPlant documentation).

[Power]PowerPlant is part of CodeWarrior, available from Metrowerks, Corp., Suite 300, 2201 Donley Drive, Austin, TX 78758.

[Pree95] W. Pree, *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[TCL] THINK Class Library is available from Symantec Corporation, 10201 Torre Ave., Cupertino, CA 95014.

[Wilson90] D.A. Wilson, L.S. Rosenstein,, and D. Shafer, *Programming with MacApp*. Addison-Wesley, 1990.

Location Independent Internet Config

Adam Treister
adam@treestar.com

Abstract

This paper discusses ideas to override components of the popular Internet Config extension so as to keep preference information on a central server, instead of the local Macintosh Preference folder. Doing so removes traditional limitations that have made it difficult to get a consistent user experience for users who move between multiple machines, or for users of shared machines, such as computer labs. The Application Configuration Access Protocol is proposed as a means to accomplish this location independence via open Internet standards.

Introduction

Internet Config has been one of the grass roots success stories of the Macintosh. It is a simple solution to an annoying problem: having to type redundant information into several programs' preference dialogs, or facing unexpected behavior because preference information is not entered consistently. It puts the user experience above any need for ownership. It has gained wide acceptance among developers, because it is easier to write a program that uses Internet Config, than to write one that doesn't.

Still, there are several situations that IC does not handle. Let's look at some of these cases (stolen verbatim from [WALL96]):

Jane is an employee of a large corporation. She works both at her office and at home after hours. She also frequently travels on business and needs to stay in touch with the home office via her laptop. She is constantly frustrated by having to reconfigure her electronic mail so that the mail she needs is available and presented in the way she wants it. She is beginning to think it's a conspiracy by tech support to make her lose productive time.

Ahmed is a home-user of the Internet via an Internet Service Provider. Ahmed's wife, Suellen, also uses their ISP account to cruise the net. Suellen and Ahmed are always fighting about the bookmarks files that they share; Suellen says that Ahmed constantly overwrites her bookmarks, and Ahmed maintains he never knows which preferences file is which because Suellen can't name her files consistently. They seem to be heading either for separate PCs and ISP accounts or divorce -- maybe both.

Hamida is a researcher for a pharmaceutical company. She has a PC in her office and a Unix workstation in her lab. She is constantly losing track of the email addresses of her colleagues, because she can't remember which electronic address book she stored it on -- the PC or the workstation. The cure for cancer is delayed every time she has to look up the same address twice.

Hector is an undergraduate student at a large research university. He goes from his dorm computer to a public lab to the library to a research lab to his friend's house in the course of his day, and wants to read his NetNews subscription to alt.music.reallyweird several times a day to keep up with the latest. The problem is he has to renew his subscription to the newsgroup at every stop along the way, because some use a .newsrc file, some use a preferences file, and some systems are used by other people who strangely aren't interested in alt.music.reallyweird and keep deleting Hector's subscription. He's thinking about switching to Barry Manilow if staying hip is this much trouble.

Prunella is a tele-commuter who works from home using her PC three days a week while she takes care of her new baby, and works in the office the other two days. Her spelling dictionary at work is on a LAN and contains a list of special words to flag in her word processor. She has to periodically load up 3 floppies of data to transfer to her PC at home so she can use the dictionary at home, and frequently misses new keywords added in by the boss during her spells at home. She's beginning to wonder if the hassle is worth the salary and if the kid really needs to go to college.

These are variations on the problem that Internet Config solves, but they establish that IC does not carry the solution far enough. IC largely solves the problem of interoperability between Mac applications, but it does little to solve the problems of platform or location independence. It reflects a classic Macintosh myopia; it assumes that users will always sit down at the same computer, and that it is a Mac.

As the market share of the Mac decreases, this is an increasingly dangerous assumption. Even the most loyal Mac users sometimes have jobs, and sometimes that means they can't use the platform at work that they prefer, or that they use at home. This doesn't diminish their desire to share address books or bookmark files. The era of the personal computer is giving way to the era of the network computer, and this is a classic example of how characteristics that once freed us from the tyranny of MIS managers, now shackle us to our own singular workspace.

In the education market, the largest niche where the Macintosh is a major platform, this situation is the rule not the exception. How many users in a K-12 classroom have exclusive use of a computer, or even use the same computer from one day to the next? Apparently the system architects at Apple think these users don't warrant a personalized environment. The design of the OS would imply that they think it's okay that students send email messages with a classmate's address in the FROM field.

Different Scenarios of Preference Management

Fortunately, the solution to all these problems is not very complicated. The Preferences folder needs to live on a server somewhere. It should be possible to get and set application options remotely. These fields should be platform and location independent.

Actually, it is something of an oversimplification to say "preferences should live on a server." In fact, where they live is a function of both the user's habits and the machine's usage profile. You need to look at the user's mobility, and at the machine's usage. If the user only uses a computer from one location, and that computer is only used by the one user, then the current model works fine.

There are a small number of possible scenarios that dictate how preferences should be stored. Figure 1, below, shows the two by two matrix representing options for configuration management, with one axis describing the user, and the other describing the machine.

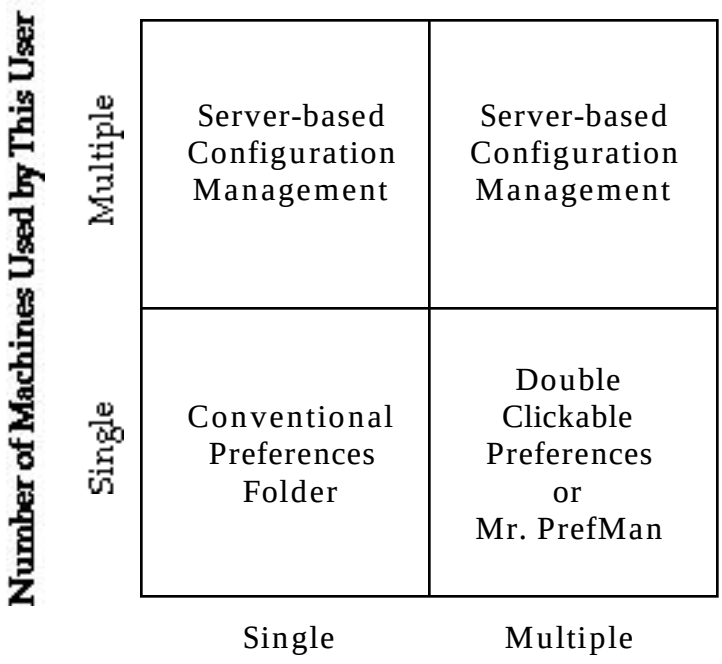

Figure 1. Configuration Matrix

The bottom left box represents the one user - one machine configuration that is currently supported in the MacOS. If you own a machine, always work there, and have physical security (or don't require security) and don't share it with others, then the current model of storing preferences in the system folder is perfectly adequate.

For the case of multiple users on the same machine, but where the users only use that one computer (bottom-right), there needs to be a method to keep multiple preference sets on a single computer. The includes many home machines, as well as small offices where the computer is a shared resource that lives in a common space, instead of on an individual's desk.

For the sake of these cases, I advocate writing applications so that the preferences file behaves more like a document. You should be able to launch the application by double-clicking on a preference file, and you should be able to change the current preferences by opening a preference file from the disk. Note that this is in direct contradiction to the Interface gurus at Apple, who have gone so far as to recommend the text in the dialog box that you should bring up when refusing to launch your application in response to a double click on the preference file. [See Woodcock, Gary, The Right Way To Implement Preference Files, develop 18]. If the application supports this feature, the machine can be set up with several preference files, each in the personal folders of the individual users.

Another solution to the problem of shared machines, and stationary users, is an extension called Mr. PrefMan, written by Quinn in response to people asking him for a way to get Internet Config to support multiple users. This extension swaps entire sets of preferences in and out of the Preference folder. It has the advantage of not requiring any modification to the existing applications on the machine.

This still is not much of a solution for the itinerant user. Depending on the market for your product, this might be someone who works from specific locations, like home and office(s). I'm going to approach this from the perspective of writing products for education, where multiple locations is the rule, not the exception. K-12 students work almost exclusively from shared classroom computers. Sometimes they add a home computer to the configuration. To make things even more interesting, the school computers tend to be Apples, and the home machines tend to be Wintel. In higher education, students often own their own computer, but they use dorm clusters and machines in the library with high frequency. They can never assume that data left on any machine is either private or persistent.

Application Configuration Access Protocol

To solve this problem, there is a new standard under development, called the Application Configuration Access Protocol, or ACAP. [NEWMAN97] This protocol has roots at Carnegie Mellon University, and comes from their transition from a proprietary mail system (the Andrew Messaging System) to an open standards approach based on IMAP4. In the process of moving, it became clear that there are features of a mail system which are not mail, but which need to be handled in a location-independent, platform-independent, scalable architecture.

The fundamental unit of storage in ACAP is a *dataset*, a named set of entries. ACAP defines a hierarchical namespace for datasets. A dataset is a set of entries; each entry is a set of attribute/value pairs. There is a *name* attribute to uniquely distinguish an entry within a dataset. The server maintains a *modtime* attribute containing the time last modified of the entry.

Dataset types may be defined and extended as needed. The initial protocol pre-defines a common set of dataset types: lists, mailbox lists, options, addressbooks, media types, and bookmarks (URLs).

Access to datasets and dataset entries are regulated by Access Control Lists (ACLs - pronounced "*ackels*") which are taken from the Andrew File System that is used widely throughout the Internet. This is similar to the Unix permission bits (controlled by chmod), but with additional options, and a finer granularity. Individual fields can be set to govern where the attribute can be read, written, deleted, listed, administered (i.e., edit the ACLs), etc. and permissions can be added and subtracted, providing a broad control over who can do what.

ACAP has a very powerful inheritance mechanism, that lets a system administrator set up default settings for new users, which they can overwrite as they customize their own configuration. As customizations are removed they revert to the inherited set. With this mechanism there can be enterprise, department, group and individual versions of data such as address books, and a new user added to the system will immediately get a good default configuration with no work on the part of the administrator. Because the success of a distributed computing architecture is measured by how cheaply you can add the nth user to the system, the ability to inherit a reasonable set of defaults is very important.

This protocol is very much still in progress. ACAP is on the IETF standards track. At the time of this writing, it is suffering the fate of many candidate standards: the chaos of democracy on the Internet. It is being pummeled with new features and new interpretations, and requests to shoehorn yet another bit of functionality into its swelling specification. The ideas expressed are individually valid, yet as the protocol attempts to accommodate them, it begins to sink under the weight of the accumulated complexity. Fortunately, clients can ignore parts of the protocol that they don't need. And for the purpose of extending Internet Config to be location independent, we need very little.

Below is a sample telnet session, that shows how one might save and restore preferences using ACAP. Commands sent by the client are shown in bold, and with a right pointing arrow. Replies sent by the server are shown in plain text, and with a left pointing arrow. The arrows and line breaks are not part of the actual protocol stream.

-> **telnet acap.treestar.com 674**
<- * ACAP IMPLEMENTATION("Chris's fingers v1.0")
-> **A001 AUTHENTICATE XXX YYYY**
<- A002 OK "Hi dude!"
-> **A002**
STORE("/options/user/<username>/vendor/TreeStar/Mailstrom/Name"
"value" "Adam Treister")
<- A002 OK "Stored for posterity"
-> **A003 STORE**
("/options/user/<username>/vendor/TreeStar/Mailstrom/Email"
"value" "adam@treestar.com")
<- A003 OK "Stored for posterity"
-> **A004 STORE**
("/options/user/<username>/vendor/TreeStar/Mailstrom/
Host.IMAP" "value" "imap.treestar.com")
("/options/user/<username>/vendor/TreeStar/Mailstrom/Host.SMTP"
"value" "smtp.treestar.com")
<- A004 OK "Stored for posterity"
-> **A005 SEARCH**
"/options/user/<username>/vendor/TreeStar/Mailstrom/"
RETURN ("value") ALL
<- A005 ENTRY "Name" "Adam Treister"
<- A005 ENTRY "Email" "adam@treestar.com"
<- A005 ENTRY "Host.IMAP" "imap.treestar.com"
<- A005 ENTRY "Host.SMTP" "smtp.treestar.com"
<- A005 MODTIME "19970527191332"
<- A005 OK "That's the stuff, man."
-> **A006 LOGOUT**
<- * BYE "Have a fabulous day"
<- A006 OK "dokay"
<connection closes>

Transcript 1. Sample Telnet Session to an ACAP server

This session shows how a simple set of STORE and SEARCH calls can read and write the entries in a dataset. The protocol supports a rich set of optional parameters to request the server to sort entries, to return only those modified since the last request, to govern how deep in

the namespace hierarchy a search may dig, etc. For a full discussion of the options, the reader should download the current draft specification from http://andrew2.andrew.cmu.edu/cyrus/acap/acap-draft-003.txt

An Interface to Internet Config

The examples below show only a minimal subset of functionality that will be available with ACAP. Because this is meant to override IC's ReadPreferences and SavePreferences functions, these are synchronous, brute-force transfer of all preferences stored in Internet Config knows to ACAP, or all entries in an ACAP dataset to Internet Config. An application that chooses to use ACAP directly for a more complex set of configuration options, such as active stocks being tracked by the application, or user dictionaries for a spell checker, would need to use more sophisticated, asynchronous context tracking. Since that is clearly more complex, it is left as an exercise for the reader.

I recommend the IC files included with John Norstad's Newswatcher [NORSTAD95] application as a starting point for implementing an application that is Internet Config aware. It contains reliable and well documented libraries that demonstrate how to use IC in a real world example application.

```
// ---------------------------------------------------------------------------
//     IC2ACAP:Extract data from Internet Config and send to an ACAP server
// ---------------------------------------------------------------------------
OSErr IC2ACAP(char* inHost,char* inUser,char* inPassword, char* inDataset)
{
       extern ICInstance gInst;
       const long kMaxBufferSize = 2000;   // profusely ugly hardcode
       char buffer[kMaxBufferSize];
       ICError icErr;
       ICAttr attr;
       Str255 key;
       icErr = ICBegin(gInst, icReadOnlyPerm);
       if (icErr != noErr) return icErr;
       long count;
       icErr = ICCountPref(gInst, &count);
       if (icErr != noErr) return icErr;
       long size = kMaxBufferSize;
       long i;
       ACAPStream acapStream(inHost, inUser, inPassword);

       for (i = 1; i <= count; i++)        // ah, Pascal
       {
              icErr = ICGetIndPref(gInst, i, attribute);
              if (!icErr)
              {
                     icErr = ICGetPref(gInst,key, &attr, buffer, &size);
                     if (icErr == noErr)
                     {
                            tag = acapStream.GenSym();
                            acapStream.Store(tag,inDataset,attribute, value);
                     }
              }
       }
       icErr = ICEnd(gInst);
       return icErr;
}
```

Listing 1. Saving Data From Internet Config To ACAP

```
// ---------------------------------------------------------------------------
//     ACAP2IC:      Suck a dataset off the net, and populate Internet Config
// ---------------------------------------------------------------------------
OSErr ACAP2IC(char* inHost,char* inUser,char* inPassword, char* inDataset)
{
       extern ICInstance gInst;
       const long kMaxBufferSize = 2000;   // profusely ugly hardcode
       char attribute[kMaxBufferSize], value[kMaxBufferSize];
       ICError icErr = ICBegin(gInst, icReadOnlyPerm);
       if (icErr != noErr) return icErr;
       ACAPStream acapStream(inHost, inUser, inPassword);

       tag = acapStream.GenSym();
       acapStream.SendSearch(tag, inDataset, "*", kAllAttributes);
       Boolean done = false;
       while (err == noErr && !done)
       {
              err = acapStream.GetResponseLine(attribute, value);
              size = strlen(value);
              done = (size == 0);
              if (!err && !done)
                     err = ICSetPref(gInst,attribute, &icFlag, value, size);
       }

       icErr = ICEnd(gInst);

       return icErr;
}
```

Listing 2. Downloading Data from ACAP and giving it to IC

The beauty of this scheme is that it is a freebie to developers. Programs that talk to the Internet Config API already offer users a unified view of those preferences which make sense to be shared between applications. By moving this information out of the Preferences folder, and onto a central server, the user gains the ability to move around to different platforms and different locations, and not lose bookmarks, address books or personal information. Users can share machines without stomping each others' preferences. Developers don't have to do anything to take advantage of the added functionality.

Apple has yet to announce what will happen to the Preference folder and user configuration information under Rhapsody. In the past, NeXT has done the right thing, but in the wrong way. They used NetDB to store the data on the server using a proprietary protocol. That's a decent solution, if you have a NeXT computer in each home and office, but is not as interoperable as it could be. Hopefully Apple can take this opportunity to move this scheme to an open standard, and offer the same benefits to users who use mixed systems and have to manage their own systems. The real winners will be administrators who have to manage thousands of diverse systems for an enterprise.

What we want is the transparent, location-independent access to configuration information that we associate with Unix, the look and feel of a Mac, and an open standard governing it all, so that when the Wintel world copies it, it will all be interoperable.

Bibliography

[NEWMAN97] Newman, C. & Meyer, J.G. *ACAP - Application Configuration Access Protocol, Internet Draft*. http://andrew2.andrew.cmu.edu/cyrus/acap/acap-draft-003.txt, 1997

[NORSTAD95] Norstad, J. *Newswatcher*, http://charlotte.acns.nwu.edu/jln/progs.html, Northwestern University, 1995

[QUINN95] Quinn. *Implementing Shared Internet Preferences With Internet Config*, develop 23, 1995

[WALL96] Wall, Matthew. *"The Application Configuration Access Protocol and User Mobility on the Internet"*, http://andrew2.andrew.cmu.edu/cyrus/acap/acap-white-paper.html2, 1996

[WOODCOCK94] Woodcock, Gary, *The Right Way To Implement Preference Files* , develop 18, 1994

QuickDraw Gems

©1997 by Gavriel State
gav@magmacom.com

Abstract

This paper describes five graphics techniques for getting the most out of QuickDraw. The gems described show: how to draw dashed lines at any angle; how to use the patXor pen mode in color to display a color marquee; how to combine multiple polygons into a single PolyPolygon; how to speed up the drawing of large numbers of polygons at once; and how to use Xor drawing modes to overcome Region complexity limitations.

Introduction

Every programmer has a bag of tricks that is collected over the course of their career. Quite often, these tricks and techniques are shared; they are published in books, in technotes, and in papers like this one. This paper describes a number of the graphics techniques that I have recently found useful; some of them are my own, while others were developed with the help of friends and colleagues. Most of the Gems (borrowing the name from the famous Graphics Gems series of books) are QuickDraw specific - the basic concepts behind them are not generally applicable to other systems, unless those systems were designed with limitations similar to those found in QuickDraw.

Dashed Lines at Any Angle

Unlike many other graphics systems, QuickDraw does not provide any direct means of drawing dotted or dashed lines. Applications that need to draw dashed lines (in order to provide users with feedback about selections, for example), usually use a special pen pattern with diagonal stripes in order to simulate a dotted pen. When a vertical or horizontal line is drawn with this pen pattern set, a dashed pattern will appear. See Figure 1(A) for an example.

This approach works fine for applications that require only horizontal and vertical dashes, but some programs need to be able to draw dashed lines at arbitrary angles. For example, a program might use dashed bezier curves to allow users to edit a shape-distortion envelope. The bezier curve would be broken up into a series of connected line segments forming a polygon, and each line segment would be drawn as a dashed line. If only one pattern were used, then the dash length would increase as the angle of the line segment approaches the pattern's diagonal angle. Note the appearance of the 45° line in Figure 1(A).

This problem can be circumvented by using multiple patterns with different dash angles, and selecting the pattern with the angle that is most perpendicular to the line we wish to draw. In its simplest form of this solution, only two patterns are used, each at opposite diagonals (though a vertical and a horizontal pattern could also be used). All lines in the upper-left and lower-right quadrant use one pattern, while the lines in the lower-left and upper-right quadrants use the other. See Figure 1(B).

If we like, we can increase the pattern resolution. The more patterns we use, the closer to perfect the dashes will be. Notice the difference between the thick vertical and horizontal lines in Figure 1(B) and 1(C). In Figure 1(B), the dash-strokes are quite clearly angled, while in Figure 1(C) they are perfectly straight.

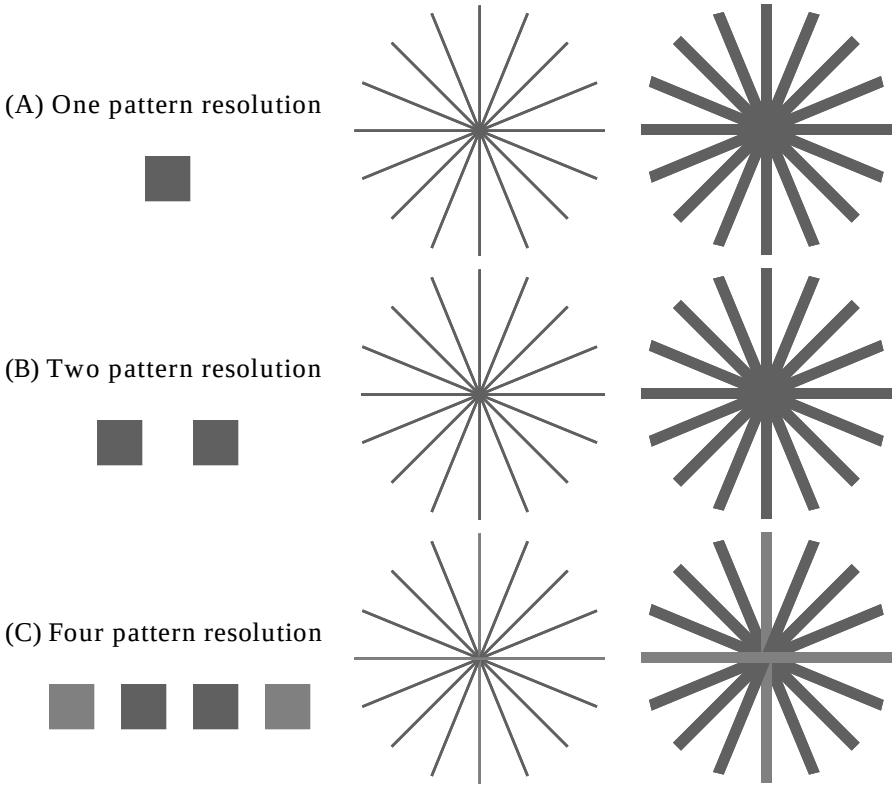

Figure 1. Angled dotted lines with varying pattern resolutions

Drawing In Color With patXor

Many applications use QuickDraw's patXor pen mode in order to produce effects like the famous 'marching-ant' marquee selection. This effect is created by setting the QuickDraw pen mode to patXor, and then repeatedly calling FrameRect() to draw a rectangle, cycling the pen pattern through a set of eight diagonally striped patterns each offset from the last by a single pixel.

Now, what happens if one wants to draw that marquee in a color other than black and white? The first thought might be to simply set the pen foreground color to the desired color and continue using the same set of monochrome patterns that were used before. After all, when the pen foreground or background color is set, and shapes are drawn using a monochrome pattern in patCopy mode, the foreground and background patterns are applied to the pattern. Why should this not also be the case in patXor mode?

As may be expected by the simple fact that there is a gem worthy of being printed here, this turns out not to be the case. QuickDraw does not perform this kind of colorization when drawing in patXor mode.

So, how can a color marquee be displayed? Two methods are available. The first, shown in [APPLE94], is to create an offscreen GWorld to hold a copy of the window area underneath the selection, and to use CopyBits to restore the pixels underneath the selection rectangle whenever

required. With this method, the marquee is drawn using the patCopy mode, so the background color is never seen behind the marquee. We can do much better than this.

The key to this problem is the 'pat' part of patXor. Rather than providing a monochrome pattern and setting the foreground and background colors, we can instead provide a *color* pattern - a PixPat. If we provide QuickDraw with a PixPat instead of a classic monochrome pattern, it respects the colors in the PixPat and properly performs the Xor operation. PixPats can be created in a resource editor and stored in a 'ppat' resource until needed, but doing this means that the marquee colors are fixed at whatever choices were made when the 'ppat' was saved.

Instead of loading a PixPats out of the resource fork, we can actually create them ourselves by transforming a monochrome pattern into a PixPat. This is possible because QuickDraw automatically converts a monochrome pattern into a monochrome PixPat when the PenPat() routine is called. Listing 2 shows how to create a PixPat from a monochrome pattern and a givn foreground color.

```
// This function converts a classic QuickDraw B&W pattern into a
// PixPat with the passed color being the foreground color.  It
// assumes that the CGrafPort we're going to draw into is the current
// port.  You must allocate the new PixPat using QuickDraw's
// ::NewPixPat() function before calling this routine.
// Note that once you have allocated the PixPat you are responsible for
// disposing of it (after QuickDraw is done using it, of course).

void ConvertPatternToPixPat(Pattern *thePat, RGBColor *foreColor,
                                PixPatHandle newPixPat)
{
       // Set the B&W pattern as the Pen Pattern.  QuickDraw will create an
       // equivalent PixPat and store it in the current port's pnPixPat field.
       ::PenPat (thePat);

       // Copy QuickDraw's automatically generated PixPat into a new
       // PixPatHandle.  Calling CopyPixPat does a deep copy on all the
       // data associated with the PixPat.
       PixPatHandle curPixPat = ((CGrafPort *)qd.thePort)->pnPixPat;
       ::CopyPixPat (curPixPat, newPixPat);

       // Now, we change the new PixPat's pattern type to make it a full
       // color PixPat instead of a simple B&W pattern.
       newPixPat[0]->patType = 1;

       // Now we modify the PixPat's color table to insert the foreground
       // and background colors.  We take the background color set in the
       // current port.
       CTabHandle theCTab = newPixPat[0]->patMap[0]->pmTable;
       (*theCTab)->ctTable[0].rgb = ((CGrafPort *)qd.thePort)->rgbBkColor;
       (*theCTab)->ctTable[1].rgb = *foreColor;

       // We now report to QuickDraw the fact that we've modified
       // some of its exposed data structures so that QuickDraw can
       // invalidate any information it has cached internally
       ::CTabChanged (theCTab);
       ::PixPatChanged (newPixPat);
}
```

Listing 2. Pattern to PixMap conversion

With this function available, all that's needed to use Xor mode in color is to set the port's foreground pattern using the PenPixPat() routine, and to set the pen mode to patXor.

PolyPolygons - How to Draw a Doughnut

Standard QuickDraw polygons are a simple collection of lines that form a closed shape (the last point in the polygon is implicitly the first point in the polygon). These polygons can be used in all the usual ways - they can be filled or stroked, and they can become the boundary of a region. One useful feature provided by many graphics systems other than QuickDraw is the ability for a polygon to be composed of a collection of both line and move operations, instead of a collection of lines only. Such a polygon is effectively composed of multiple sub-polygons, and because of this these structures are sometimes called 'PolyPolygons' on systems that have this feature. Due to the standard even-odd fill rule used in most systems, PolyPolygons can be hollow in one place and filled in others.

This feature is useful in a number of circumstances. For example: when drawing a filled shape with a hole in it - such as a flattened torus - PolyPolygons can be used to avoid the need for a clipping region. Figure 3 contrasts how a doughnut shape might be drawn using a clipping region, as opposed to using a PolyPolygon.

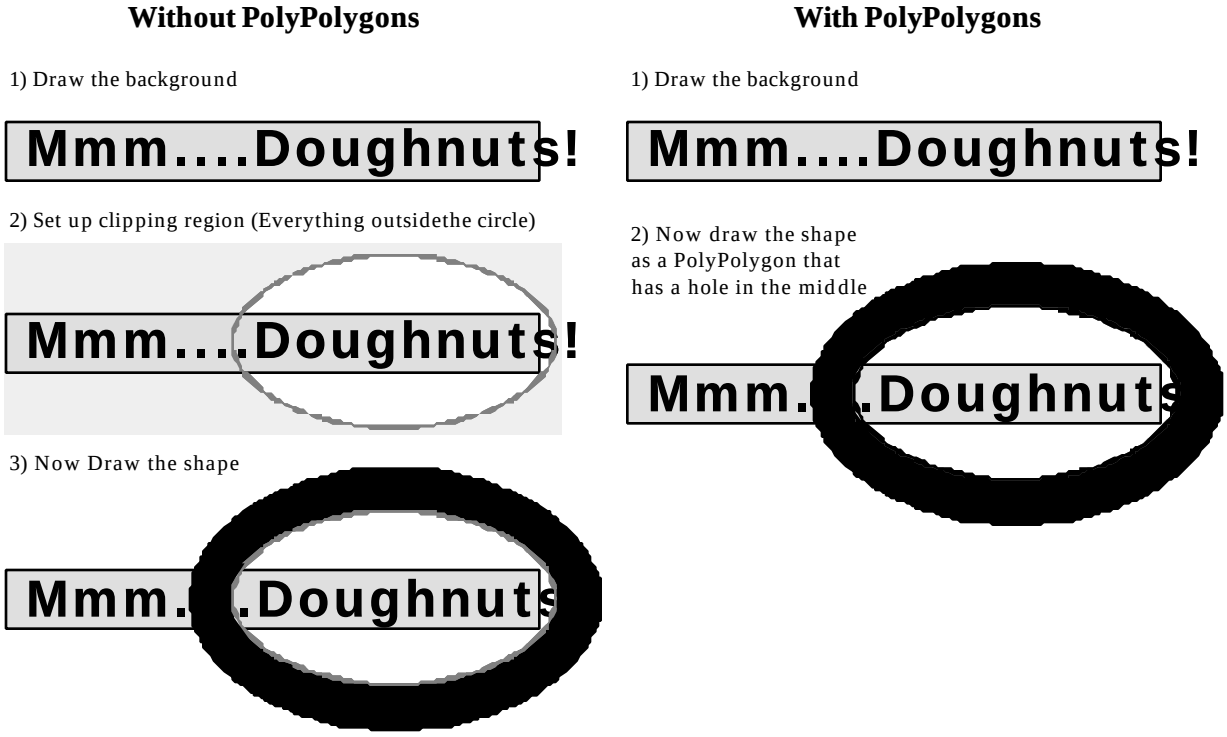

Figure 3. Drawing a doughnut - with and without PolyPolygons.

Without PolyPolygons, we need to remove the inner (hollow) part of the shape from the active clipping region after drawing the background, but before drawing the filled part of the shape. Setting up a clipping region like this can be expensive - QuickDraw has to perform complicated calculations to determine the new region. On top of this, QuickDraw's Region structures can overflow if they are too complex (over 64K), and they are not available during PostScript printing.

When PolyPolygons are used instead of clipping, we can completely avoid dealing with QuickDraw Regions. Simply rasterizing the PolyPolygon gives the desired result, which is also printable on any printing device.

How do we go about creating a PolyPolygon? It's actually quite simple. In systems that directly support PolyPolygons, a PolyPolygons is a structure that records the equivalents of both the MoveTo() and LineTo() functions in QuickDraw. Since QuickDraw supports only LineTo() calls in its Polygon structure we need to add a fake MoveTo() call using a combination of LineTo() calls. We do this by inserting a pair of overlapping LineTo() calls into the QuickDraw Polygon structure: one when moving to each sub-polygon, and the other when popping back to the previous sub-polygon. Figure 4 shows a simple square PolyPolygon.

```
MoveTo   (10,  10);  // Start

LineTo   (110, 10);  // 1st sub-poly
LineTo   (110, 110);
LineTo   (10,  110);
LineTo   (10,  10);

LineTo   (20,  20);  // Move in

LineTo   (100, 20);  // 2nd sub-poly
LineTo   (100, 100);
LineTo   (20,  100);
LineTo   (20,  20);

LineTo   (10,  10);  // Pop to start.
                     // Finished
```

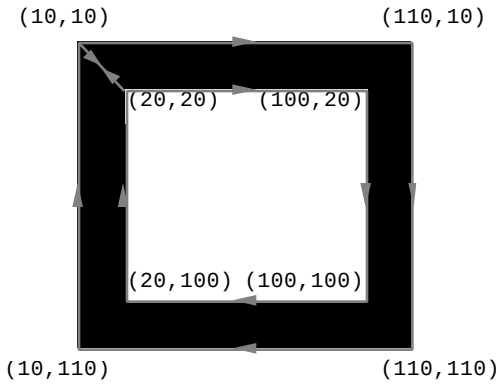

Figure 4. QuickDraw commands for an example PolyPolygon:

Since QuickDraw always uses an even-odd fill rule, the filled area of any PolyPolygon is essentially the area of each sub-polygon Xor-ed in sequence. This filled area can be filled using PaintPoly() or FillPoly(), and it can be made into a QuickDraw Region in the usual way. It is important to note, however, that due to the way that we have constructed our PolyPolygon, FramePoly() should not be called with a PolyPolygon. QuickDraw's FramePoly() routine does nothing more than walk through the list of points in a Polygon, calling LineTo() as it goes; with a PolyPolygon constructed as above, this would result in the lines connecting sub-polygons being drawn. If drawing the outline of the PolyPolygon is required, it's best to make a direct sequence of LineTo() calls for each sub-polygon.

Speedygons - Drawing Polygons More Quickly

In QuickDraw, as with any graphics system, the programmer must supply a set of points that specify a polygon's geometry before that polygon is drawn. The mechanism provided by QuickDraw to set these points is very simple: a call to OpenPoly(), followed by a MoveTo() call and a set of LineTo() calls, topped off with a call to ClosePoly(). When OpenPoly() is called, a new PolyHandle - a Handle to a Polygon structure - is allocated in the application heap, and this PolyHandle is automatically grown as new points are added to it. When done with the polygon, the programmer calls KillPoly(), and the memory occupied by the Polygon is deallocated from the heap.

This method is sufficient for drawing the odd polygon here and there, but if a program needs to draw thousands of polygons at once (as is often the case when drawing complex vector illustrations), the overhead for repeated memory allocation and deallocation can become a significant penalty. This will be especially true if many small polygons are drawn - the ratio of time spent drawing to the time spent managing memory decreases proportionally with the number of pixels that need to be drawn. There are a number of ways of improving performance in this situation.

The first, and often the fastest, is to allocate an off-screen GWorld and bypass QuickDraw's polygon rendering altogether. This is a good solution, but can require a significant investment in development time depending on what features of QuickDraw need to be emulated in the

offscreen drawing. Another problem with this solution is that by bypassing QuickDraw, one can no longer record a drawing into a PICT, except as a bitmap. Additionally, the benefit of any graphics acceleration hardware that the user might have is lost.

Another way to speed up drawing is to take note of the fact that the internal layout of the QuickDraw Polygon structure has been available in Inside Macintosh since its first publication. Listing 5 shows this structure.

```
struct Polygon {
       short         polySize;
       Rect          polyBBox;
       Point         polyPoints[1];
};
typedef struct Polygon Polygon;
```

Listing 5. QuickDraw's Polygon structure

Rather than using OpenPoly() to allocate a PolyHandle for each polygon, we can allocate a single handle large enough to hold the most complex polygon that is to be drawn. When drawing, we set the points directly in the pre-allocated polygon before calling PaintPoly() or FillPoly(). While setting the points, we need to calculate the polyBBox field by taking the minimum x and y values of the points we are adding as the left and top of the polyBBox, and the maximum x and y values as the right and bottom. The polySize field must also be adjusted, and it's important to note that this field measures the number of bytes in the structure, not the number of points.

By bypassing the normal way of specifying the points in a polygon, we not only save on memory allocation overhead, but also on function call overhead, since we no longer have to use a LineTo() call to add a new point to a polygon. Function call overhead for Toolbox routines is significant on 68K machines due to the trap dispatch mechanism, and even more so on the PowerPC machines due to the fact that toolbox calls are cross-TOC, and that the trap dispatch mechanism has to go through the Mixed Mode Manager in case a 68K extension has added patched a trap.

What kind of performance improvement does this optimization give? In certain tests, improvements of up to 35% have been observed, but as discussed above the results will vary with the average size of the polygons that are drawn.

There are a few important caveats to keep in mind when using this method. Despite the fact that the Polygon structure has been public knowledge since 1984, some Apple engineers (though no tech notes) have warned against this technique, claiming that graphics accelerators may be confused by the 'artificial' polygons if they perform any caching based on the PolyHandles they've been passed. This line of reasoning doesn't hold water for two reasons: first, it is quite rare for a program to need to draw the exact same polygon over and over again in a tight loop, thus making polygon caching relatively worthless; secondly, Apple has provided special functions to notify QuickDraw of direct manipulations of many other data structures (ie: GDeviceChanged(), etc.) - if it is important for QuickDraw to know when a Polygon structure has changed, a similar notification routine should exist for PolyHandles.

Unfortunately, the warnings about direct manipulation of the Polygon data need to be taken seriously - there are at least three problems with some system-level software related to the use of this technique. The first is that QuickDraw will sometimes mark an artificial PolyHandle as purgeable if it is drawn during the recording of a PICT. The solution for this problem is to save and restore the Handle state before calling the PaintPoly() or FillPoly() functions. Another problem occurs when using graphics acceleration with a PCI Twin Turbo 128 video card: triangular patterns will be drawn through polygons drawn in this manner.

One way to avoid being bitten by these potential problems, while still improving the speed of graphics code is to give the user the option to choose what levels of compatibility vs. performance they would prefer.

Complex Clipping Using Xor

QuickDraw's Regions are powerful tools for graphics programming. Regions can be combined with one another in a variety of different ways, they can be filled or painted, they can be outlined, and last but not least, they can be used to clip out subsequent drawing operations - confining them to those areas that are within the visible and clipping Regions of a specified GrafPort.

The internal structure of QuickDraw Regions encodes the horizontal boundary lines that border areas within the region. This encoding is very efficient for combining shapes with many purely horizontal and vertical edges, but it is much less efficient when diagonals or curves are required. In these cases, the region format effectively degenerates into a simple run-length encoded bitmap, where different horizontal boundary information is stored for each scan line in the region. Further information on the internal format of Regions can be found in [DUBIN88].

This encoding format would not be a limiting factor for the purposes of clipping if it wasn't for the fact that there is a severe limitation on the maximum size of a Region. Due to design considerations on the original 68000-based Macintosh, Region data is limited to a maximum of 64K bytes, including header information. In today's world of very high resolution displays, this limit is insufficient for representing regions that can result from combinations of large, complex polygons, or text.

Luckily, however, there is a way to clip objects to regions more complex than those that QuickDraw is capable of. The solution is not ideal - it requires that the graphics drawn within the clip region be drawn multiple times, in an Xor graphics mode - but it does provide an alternative when QuickDraw is incapable of representing a region.

The procedure for Xor clipping is straightforward. First, we use Xor mode to draw whatever graphics we want clipped. Note that it's best to draw only a single shape or bitmap; overlapping shapes can be problematic. If we must draw something complex, we draw it into an offscreen bitmap first (in a Copy mode, not Xor mode), then we use CopyBits in srcXor mode to draw the bitmap into the destination port. We refer to the graphic being clipped as the 'source' image.

Once this is accomplished, we set the pen mode back to patCopy, then we draw the clipping shape in the Xor base color of the destination GrafPort. When a pixel value is Xored with the Xor base color, the resulting pixel value is the same as the original. For 16 and 32 bit true color ports, the Xor base color is always pure white. For indexed-color ports, this value is always color table index zero, since for these ports QuickDraw's Xor mode is simply a direct bitwise Xor of the destination port's pixel values with the source. If we are drawing to the screen, we need

to iterate through the GDevice table to choose the appropriate Xor base color for each physical device the port intersects.

By drawing the clipping region in the Xor base color we pave the way for the final pass, this time drawing the source again in Xor mode. Any pixels that were set outside the clipping area are now reset to whatever value they had before our first pass. Pixels inside the clipping area now hold the result of the Xor base color Xor-ed with the corresponding source pixels - in other words, the source pixels.

Figure 6 shows a situation where we might want to use Xor clipping to achieve our desired results. If we were to draw this image using a QuickDraw Region to clip out any area outside the text boundaries we can easily overflow the maximum allowable complexity, especially if we need to draw at a very high resolution. Also note that if we were to use QuickDraw Regions, the text 'Clip Me' would have to be converted to a Region by extracting the TrueType curves for each letter, tessellating them into polygons, and combining the polygons.

Figure 6. Desired clipping result

Figure 7 shows the Xor clipping process used to produce the result shown in Figure 6. A typical 8-bit paletted destination port with color index zero equal to pure white is assumed. First, the source image is drawn in Xor mode. Note that since the previous background was pure white, the Xor operation results in the source image being directly transferred to the destination. Next the clipping area is drawn in white, sincethat is the Xor base color for this port. The clipping area can be drawn using a simple call to DrawString() - though we would need to extract the TrueType curves if we needed to draw rotated or skewed text. Next, the source image is Xor-ed on again, resulting in the clipping effect. Finally, the PolyPolygon is outlined to allow us to see the precise borders of the clipped area more clearly.

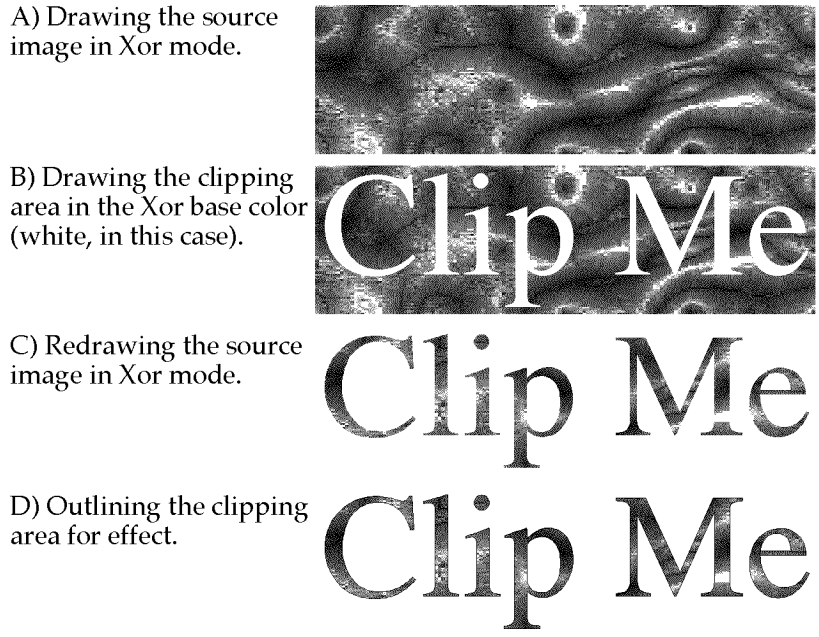

Figure 7. The Xor clipping process

Bibliography

[APPLE94] Thompson, Nick, QuickDraw Snippets - Color Marquee. 1994.
http://devworld.apple.com/dev/techsupport/source/SQuickDraw.html
[DUBIN88] Dubin, Stephen, V.M.D., Ph.D., Moore, Thomas W. , Ph.D., Fun With Regions, Part II, MacTech(MacTutor) Magazine, Volume 4, Issue 9. 1988.

Drawing to the Screen from an MP Task

© 1997 by David Slik
dslik@paradata.com

Abstract

This paper describes a mechanism to allow MP tasks to display directly to the screen. By partially overcoming the limitation of not being able to call QuickDraw, MP programmers can thus increase the functionality and capabilities of their MP software. In addition, this paper introduces MP GrafLib, a MP aware graphics library that can be extended to allow full featured drawing capabilities.

Introduction

The Multiprocessing extensions to the Macintosh Operating System, jointly developed by Apple Computer, Inc, and DayStar, Inc, open up many new possibilities for software development on the MacOS platform. By providing a runtime environment for the support of fully preemptive threads that can run on multiple processors, the way is paved for the development of scalable high performance systems, semi-realtime software and TRUE multitasking, where processes can be executed simultaneously. This run time environment and API has been successfully implemented by many developers including Adobe, Deneba and others. Unfortunately, one of the tradeoffs required to allow this level of functionality is restrictions on the type of code called. Like VBR tasks and code at interrupt time, almost all ToolBox functions can not be called. As the MP environment is intended for computational intensive tasks, this trade off is appropriate, but one key area where this limits the abilities of the developer is that there is no supported mechanism to display graphics to the screen.

This paper presents a method that allows the construction of a library of routines that allow graphic elements to be drawn directly from an MP task.

A Brief Overview of the MP Environment

An MP task can be thought of as a special function that runs as part of your program. It has several unique attributes and restrictions, and like most specialized environments, is best suited for specific applications.

MP Tasks are preemptively multitasked. This means that a task will always get processor time, but it can be stopped and started at any point. Additionally, MP tasks can be distributed across multiple processors. This allows programs using multiple MP tasks to do computation to take advantage of multiple processors and the associated performance increases. As many of the parts of the MacOS ToolBox do not support preemptive multitasking, toolbox routines can not be called from an MP task.

MP Queues

Because software running under the MacOS Runtime Environment and the MP Runtime Environment can not communicate directly, mechanism have been set up to allow executing software in different environments to interoperate. Included with the MP API is a series of calls to create and manage queues, which are used for communication and synchronization purposes. It is through these queues that parameters and other information is passed back and forth from an MP task.

Things to Remember

- PowerPC only
- Must use MP Memory management routines
- Can Not call ToolBox routines

More information about the MP Environment and API can be found in the included MP SDK, which can be found in the "Multiprocessing SDK #2" folder.

History of Low Level Graphics Display Mechanism in the MacOS

Direct to Screen

Before the advent of today's accelerated video cards, all graphic elements on the screen were drawn directly to the video memory. When a QuickDraw function was called, code inside Quickdraw would rasterize the element and set the value of all effected pixels. This mechanism, while adequate for black and white and 4 bit color, was quickly phased out after the introduction of Color QuickDraw.

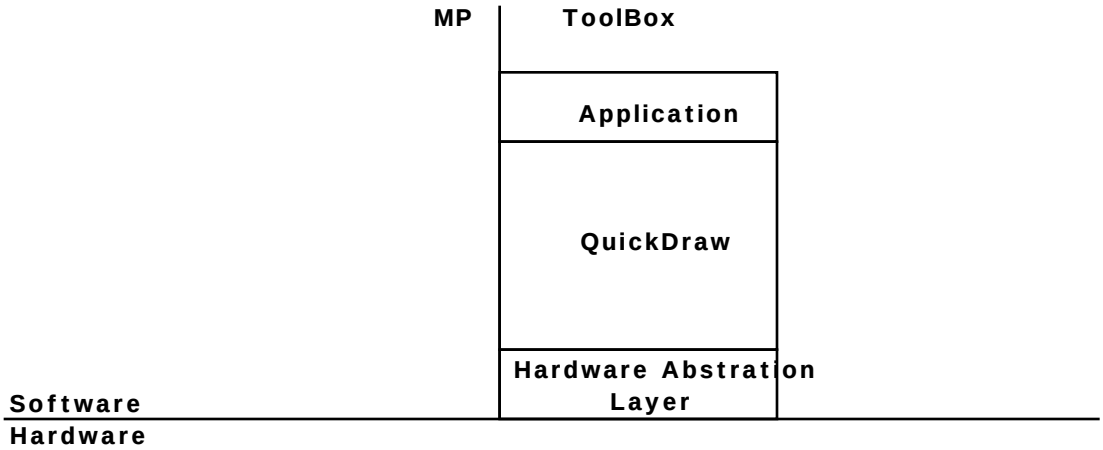

Figure 1: QuickDraw Direct Drawing

QuickDraw Acceleration

QuickDraw, and all of the Toolbox components that are built on top of it are structured around a low level library that supports access to accelerated routines within the video card. This library acts as an hardware abstraction layer that translates a QuickDraw command to draw a graphic primitive into a native command of the graphic processor on the video card. This is

how "QuickDraw Acceleration" is performed, and is the reason why graphic speeds are as fast as they are. Instead of having to fill every pixel of a rectangle, a single command can be issued to the graphics processor, leaving the main processor free while the graphics processor utilizes it's high speed local memory bus to fill in the pixels at a much faster rate then the main processor could.

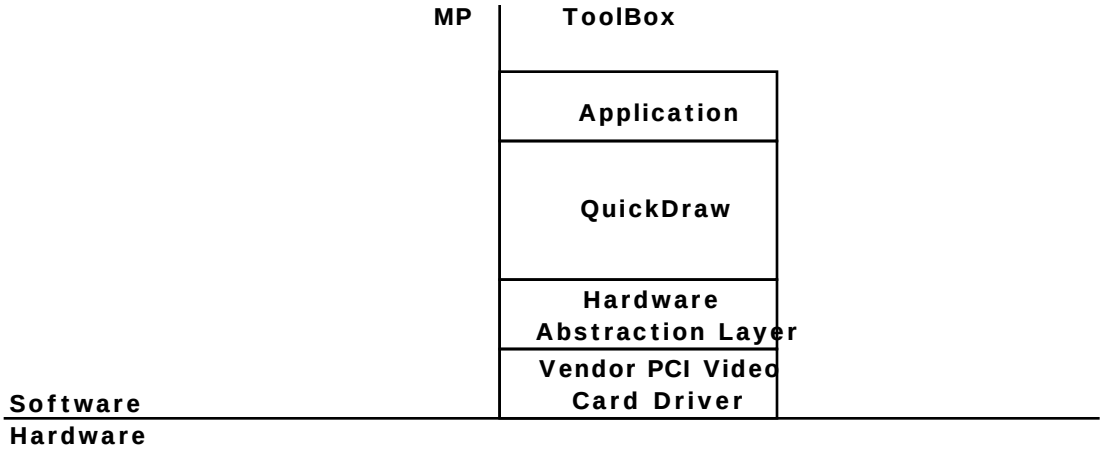

Figure 2: Accelerated QuickDraw Drawing

Mechanisms for MP drawing

In an MP task, the QuickDraw libraries are not available, and a different approach needs to be taken. Two approaches can be taken, both with different advantages and disadvantages: Direct to Screen, and Accelerated.

Direct To Screen MP Drawing is the simpler option of the two. It is straight-forward to implement, and allows elements drawn by an MP task to co-exist with graphic elements drawn by conventional tasks. This allows the MP task to draw within a MacOS window. This is the method that is focused on in this paper.

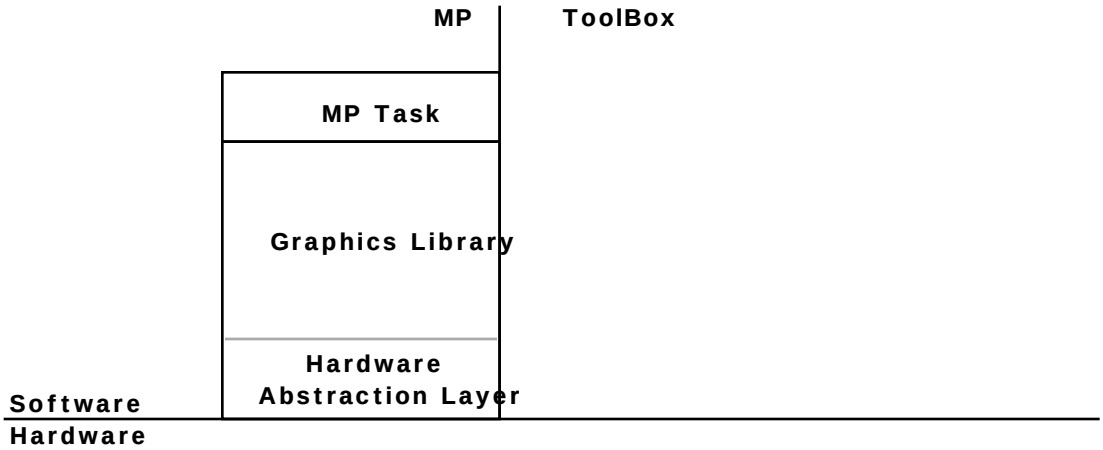

Figure 3: Direct to Screen MP Drawing

Accelerated MP Drawing requires a dedicated screen, and the development of a PCI driver. This allows the graphics processor on that PCI card to be access directly and used to draw the primitives, taking the load off the central processor. This offers many advantages, as it is much faster then direct to screen, and in many solutions, having a separate screen is beneficial. The primary disadvantage is the required complexity. To properly develop a graphic library using this approach, a layered approach would be required to ensure support for the many PCI cards that are available in the marketplace. As every graphic processor has different commands, this would be a major development effort.

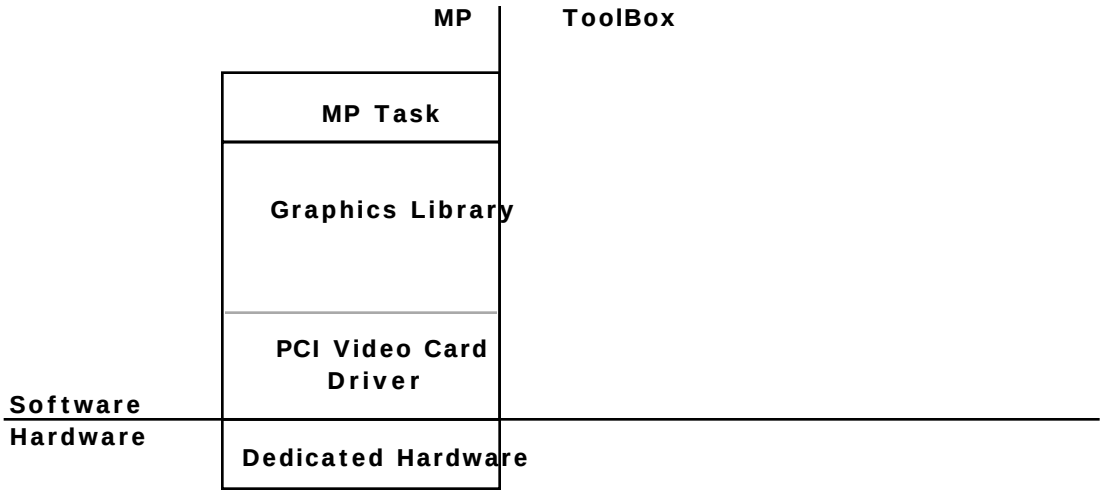

Figure 4: Accelerated MP Drawing

An Overview of Direct to Screen MP Drawing

There are three steps required to draw to the screen from an MP Task: Finding the base address of the video buffer, finding and calculating display parameters, and manipulating memory to draw.

Step #1 - Finding the base address of the video buffer

The first step required to directly access the video buffer is to find the base address.

Note that this step can not be accomplished from an MP task, and must be executed by the host application!

Before the advent of the second generation of PowerPC based systems, finding the base address of a video buffer was, at best, undocumented. With the addition of the PCI bus, a new API called the Name Registry was integrated into the system to provide a unified mechanism to access, store and retrieve hardware and driver status. By using the Name Registry, a program can discover enough information to write directly to the screen from an MP task.

The Name registry is organized in a tree structure, with devices being branches off their parent bus.

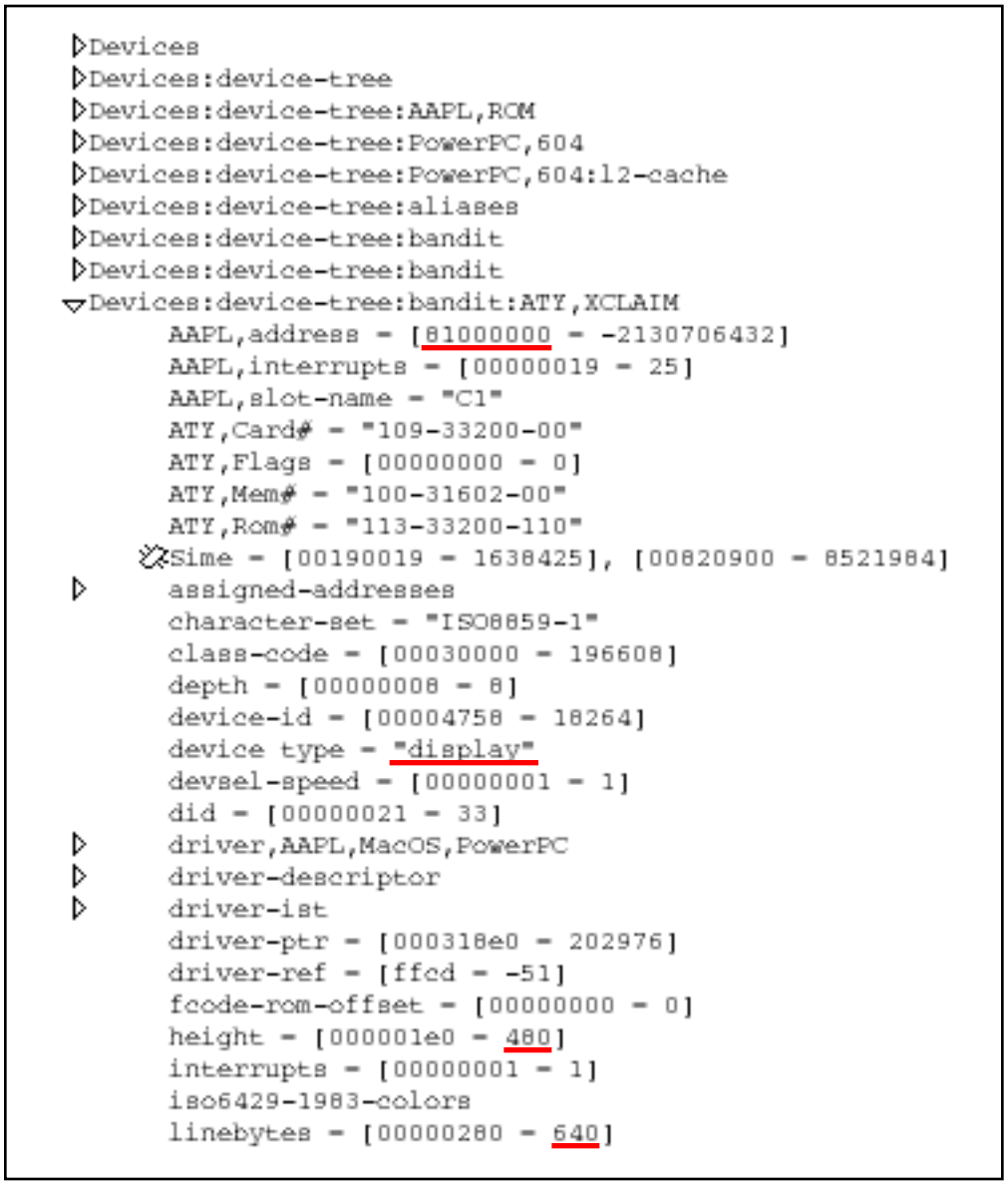

Figure 5: Name Registry Structure

Note the highlighted values in the above name registry dump. These include `APPL,address`, or the base address of the frame buffer, `device type`, which identifies the device as a display, `height`, which indicates the vertical resolution in pixels, and `linebytes`, how many bytes per row of pixels. Not shown are `width` and `depth`. The use of a constant for the device type identifier allows the name registry to be searched using several toolbox functions.

Detailed documentation and API information about the Name Registry can be found in [APPL94]

The sample code below will recurse through the name registry and call SysBeep for every display card found. A Metrowerks project for this example can be found in the (Example 1) folder with the accompanying files.

```
RegEntryIter          RegistryEntryIterator;
RegEntryID                   RegistryEntry;
RegEntryIterationOp   RegistryEntryIteratorOperator = kRegIterRoot;
Boolean               IterateDone = FALSE;
OSStatus                     theError = 0;
long                         DelayTime = 0;

RegistryEntryIDInit(&RegistryEntry);
theError = RegistryEntryIterateCreate(&RegistryEntryIterator);
if(!theError)
{
       while(!IterateDone && !theError)
       {
              theError = RegistryEntrySearch(&RegistryEntryIterator,
                     RegistryEntryIteratorOperator, &RegistryEntry,
                                   &IterateDone, "device_type", "display", 8);
              if (!IterateDone && !theError)
              {
                     SysBeep(0);
                     Delay(30, &DelayTime);
                     RegistryEntryIDDispose(&RegistryEntry);
              }
              RegistryEntryIteratorOperator = kRegIterContinue;
       }
}
RegistryEntryIterateDispose(&RegistryEntryIterator);
```

Listing 1: Beep for every video card

NOTE: the text "`device_type`" in the above source code sample may need to be replaced with the text "`device-type`" to work on computers other then the Power Macintosh 9500.

This code is used as a foundation to create the function `Ptr GetIndVideoBaseAddress(unsigned int Display);` , which recurses through every display card and returns the associated base addresses. Source code for this function can be found in the (Example 2) folder.

This function accepts a number that indicates which video card should be queried. For example, passing 1 will result in the base address of the first being returned, and so on. If a video card is not found, the function returns NULL.

Step #2 - Finding and calculating video parameters

The second step required is to find information about the selected display.

Note that this step can not be accomplished from an MP task, and must be executed by the host application!

Once the base address has been found, other parameters need to be retrieved for the selected video card. Again, this information can be retrieved using the Name Registry. The function

returns `RetreiveVideoParameters()` display information about a video card with a specific base addresses. Source code for this function can be found in the (Example 3) folder.

```
OSErr RetreiveVideoParameters(Ptr BaseAddress, unsigned int *resX, unsigned int *resY, unsigned int *BitDepth);
```

This function accepts a base address, and sets the values of resX, resY and BitDepth. If the base address is invalid, all three will be set to zero and an error indicator will be returned.

Once obtained, This information needs to be passed to an MP Task via `MPCreateTask` or `MPNotifyQueue`.

Step #3 - Setting Pixels

The third and final step required to directly access the video buffer is to set memory values to display pixels on the screen

Note that this step CAN be accomplished from both MP tasks and normal MacOS applications.

This allows the MP Library to be used from both normal and MP tasks, for development convenience.

Pixels are set by changing a segment of memory that coincides with a pixel on the screen. The address of the memory where the value is written to is calculated from the information about the display that is passed by the previous step. The size of the value which is written to set the color of the pixel is dependent on the bit depth of the screen and if direct or indexed color tables are in use.

The calculation follows as:

Memory Address = BaseAddress + (XResolution * BitDepth * Y) + X * BitDepth

Where:

MemoryAddress is not Greater then (XResolution * YResolution * BitDepth)

BitDepth is in bytes.

Once a pixel drawing routine is in place and optimized, routines to draw various graphic primitives can be built. All graphical elements used in the MacOS are composed of simple graphic elements, lines, circles, bitmaps and areas. From these basic elements, complex graphics and interfaces can be built. By manipulating pixels, all of these primitives can be rendered. Currently, lines, and rectangles are implemented, allowing a vast majority of graphics to be drawn. Algorithms for rendering various graphic primitives are widely available, and more information can be found in [FOL93].

MP GrafLib

The MP GrafLib is a collection of routines for drawing to the screen. Supporting features such as clipping, it forms an extendable platform for constructing a robust graphics library.

The example below is the GrafLib code that accompanies a program that creates a modal window at the location 100,100 that is 300 pixels high and wide.

```
DS_RecttheClippingRect;

InitMPGrafLib(BaseAddress, BitDepth, ResH, ResV);
DS_SetRect(&theClippingRect, 100, 100, 400, 400);
SetClippingRect(theClippingRect);

DS_SetForeColor(DS_SetColor(0,0,0));

DS_MoveTo(100 ,100);
DS_LineTo(500, 200);

DeInitMPGrafLib();
```

Listing 2: Simple MP GrafLib Program

This program will draw a single black line across the window.

MP GrafLib API

The MP GrafLib API is split into four distinct areas: Initialization, Color, Drawing and Utilities.

Initialization Routines

Initialization routines set up the library and initialize internal structures.

```
OSErr InitMPGrafLib(Ptr BaseAddress, short BitDepth, short HRes, short VRes);
OSErr DeInitMPGrafLib(void);
OSErr SetClippingRect(DS_Rect theClippingRect);
```

Color Routines

Color routines allow color values to be created and used when objects are drawn to the screen.

```
void DS_SetForeColor(DS_RGBColor theColor);
void DS_SetBackColor(DS_RGBColor theColor);
DS_RGBColor DS_SetColor(unsigned char Red, unsigned char Green, unsigned char Blue);
DS_RGBColor DS_GetForeColor(void);
DS_RGBColor DS_GetBackColor(void);
unsigned int DS_GetBitDepth(void);
```

Drawing Routines

Routines that allow objects to be drawn to the screen. These routines mirror equivalent ToolBox routines, with different types to prevent accidental calling of ToolBox routines. Currently they consist of functions for the drawing of pixels, lines and rectangles, both filled and outlined.

```
void DS_SetPixel(short xCooridinate, short yCooridinate, unsigned int PixelValue);
void DS_MoveTo(short horiz, short vert);
void DS_LineTo(short horiz, short vert);
void DS_Line(short distHoriz, short distVert);
void DS_FrameRect(DS_Rect *theRect);
void DS_FillRect(DS_Rect *theRect);
```

Utility Routines

Routines for common object parameter manipulation. These routines mirror equivalent ToolBox routines, with different types to prevent accidental calling of ToolBox routines. Currently they consist of Rect creation and manipulation functions for drawing rectangles.

```
void DS_SetRect(DS_Rect *theRect, short rLeft, short rTop, short rRight, short
rBottom);
void DS_OffsetRect(DS_Rect *theRect, short distHoriz, short distVert);
void DS_InsetRect(DS_Rect *theRect, short distHoriz, short distVert);
Boolean DS_RectInRect(DS_Rect *theRect, DS_Rect *InsideRect);
```

Things to do with MP GrafLib

Here are a few ideas of things to do with MP GrafLib:

- Write a MP Progress bar update routine by drawing on top of a modal dialog box
- Write visual debugging indicators for MP Tasks

Here are a few more exotic ideas of things to do with MP GrafLib:

- Write a program that keeps on drawing, even in Macsbug!
- Write a program that displays the contents of low memory in realtime to a second monitor.

  (where the color represents the memory value)

Future Additions and Improvements

As with any unfinished work, there are many improvements and additions that can be added to the MP GrafLib software. These include:

- Native MP PCI Video card drivers for accelerated drawing
- Support for additional graphic primitives (eg, arcs, circles, ovals...)
- Support for additional bit depths other then 32 bit
- Improved clipping algorithms
- Support for regions
- Support for Patterns
- BitMap support with CopyBits like functionality and offscreen buffers
- Text rendering support
- And, as always: Faster rendering speeds

In addition, once a firm foundation is in place, the addition of a windowing library, and possibly a video playback architecture would allow complete applications to be written as MP tasks.

If you have ANY comments, suggestions, corrections or additions, let me know. I can be reached at dslik@paradata.com.

Bibliography

[APPL94] Apple Computer, Inc. *Designing PCI Cards and Drivers for Power Macintosh Computers*. Apple Computer Inc, Cupertino, CA. 1996.

[FOL93] Foley, James D. *Computer Graphics: Principles and Practice.* Addison Wesley, Reading, Massachusetts. November 1994.

Enabling Your Application for Multi-byte Text

©1997 by Nat McCully
nat@claris.com

So, you have developed the next greatest widget or application and you want to distribute it on the Net. Your application has a text engine, maybe simple TextEdit, or one of the more sophisticated engines available for license, or even one you wrote yourself. One day, you get an e-mail from someone in Japan:

> Dear Mr. McCully,
> My name is Takeshi Yamamoto and I use your program,
> MagicBook, everyday. But, I have a problem using Japanese
> characters in it. When I hit the delete key, I get weird
> garbage characters. My friends and I wish to use both
> Japanese and English in your program, but it does not work
> properly. Please fix it!
> T. Yamamoto

Suddenly, there are people on the other side of the world who want to use your application in their language, and you are faced with a dilemma. You have no first-hand knowledge of the language itself, but you may be somewhat familiar with the Macintosh's ability to handle multiple languages in a single document with ease. Simply cracking open Inside Macintosh: Text seems daunting. How will these new routines affect the performance of your program? Will you introduce unwanted instability and anger your existing user base? Where can you find information on how to use these routines best, not just a description of what each routine does?

This paper will attempt to address some of these issues, and in general familiarize you, the reader, with some of the best things that make the Mac an excellent international computing environment. Intelligent use of the Macintosh's international routines, WorldScript, and the other managers in the Toolbox can be the difference between a US-only application and a truly "world-ready" tool that any user, anywhere, can utilize as soon as they download it to their hard disk. Although this paper deals primarily with Japanese language issues, the concepts outlined herein can be used with any multi-lingual environment.

What is WorldScript?

WorldScript is the set of patches to the system that enables the correct display and measurement of multi-lingual text. Over time, many of these patches have been rolled into the base system software, but even in MacOS 7.6.1, you will find a set of WorldScript extensions in the Extensions folder when you install one of the Language Kits available from Apple. The concepts and code snippets in this paper will work equally well on, for example, the Japanese localized MacOS, or on a standard U.S. system with the Japanese Language Kit (JLK). A good source of localized system software and Language Kits is the Apple Developer Mailing CD-ROM, available from the Apple Developer Catalog. WorldScript is one of the Apple-only technologies that makes multi-lingual computing possible in a far easier way than the other guys. And, when it comes to having Chinese, Korean and Japanese all in the same document, WorldScript, on MacOS, is the only thing out there.

How to Use the Script Manager and International Utilities to Make Your Application Two-byte Savvy

OK, let's get to the meat, you say. How do you make your text engine handle two-byte characters? Well, before giving you a bunch of code, let's explain how the Mac handles two-byte text.

What is a Script?

Each language that the Mac supports is grouped into categories called "scripts." For example, English and the other Roman letter-based languages like French and German all belong to the Roman script. Japanese belongs to the Japanese script. Character glyphs in the Roman script are each represented by a single-byte character code. Japanese characters are represented by a 16-bit (2 byte) character code.

Setting Up the Port — Pre-System 7 API's versus New API's

On the Mac, each font is also associated with a script. There are Roman script fonts like Helvetica and Palatino, and Japanese script fonts like Osaka and Heisei Minchou. As you know, when text is drawn into a QuickDraw `grafPort`, you first set up the port with the appropriate font, size and style, and then call `DrawText()` to draw the text. If you are using the old Script Manager API's (like `CharByte()` and `GetEnvirons()`), you need to set the port to a font in the script you are interested in using. Once the port is set to a particular font, calls to the Script Manager will follow the rules of that font's script. So the port, by way of setting the font, also has an implicit script setting. This is the key to using the Script Manager routines so they will return the correct information to your application using the older API's. The new API's have a `script` parameter, so it is not necessary to set the font of the port before using them. Since the Script Manager doesn't have to call `FontScript()` to find out the script of the current font before passing the script to WorldScript, using the newer API's could speed up your application in certain cases.

Adding Script-savvy Features to your Application

First, you need to determine if the user's system has a non-Roman script system installed. One way to find out all the scripts installed is to loop through all 33 possible script codes (`smRoman` being `0` and `smUninterp` being `32`) and calling `GetScriptManagerVariable()` with the selector `smEnabled`. Roman script is always enabled.

```
ScriptCode    script;
for (script = smRoman + 1; script <= smUninterp; script++)
{
       if (GetScriptManagerVariable(script, smEnabled))
              return TRUE;  // non-Roman script present...
}
```

Listing 1: Finding Which Scripts are Installed

Of course, simply returning `TRUE` doesn't necessarily do anything that useful, instead you could at that point initialize your internal data structures that deal with specific script systems, such as line-breaking tables, on a script-by-script basis.

Line Breaking

Most applications don't rely entirely on the Script Manager for line breaking, hit testing, or word selection, because using those routines is thought to be too slow. It is possible to optimize your text engine so that you incorporate the correct behaviors for each script system present, while maintaining the highest possible performance. The Toolbox call for finding line breaks is `StyledLineBreak()`. To use it, however, you must restrict the text you pass it to lengths of less than 32K (actually, this is true of the whole Script Manager, so tough) and text widths to whole pixel values (can you say 'rounding error?'), and if you are explicitly scaling the text, it won't work at all. You must also organize the text you pass to it in terms of script runs and style runs within them. Therefore, most applications that have word-processing functionality choose to implement their own line-breaking code that is customized for their own needs. Unfortunately, many of these private implementations break when used on WorldScript systems.

Line Breaking with Japanese Text

The simplest line-breaking algorithm for English text is to look for a space `(ASCII 0x20)` character in the line near the graphic break, and if there is none, to break on the byte-boundary nearest the graphic break. Japanese text is a bit more complicated. Japanese text has no spaces, so you must break at the character boundary nearest the graphic break. There is an additional wrinkle: Certain characters are not allowed to begin a line, and certain characters are not allowed to end a line. This set of line-breaking rules is referred to as *Kinsoku shori*. For example, you cannot begin a line with a two-byte period. You cannot end a line with a two-byte open parenthesis followed by more text on the next line. A list of *kinsoku* characters is available from the Japanese Standards Association in the form of a Japanese Industrial Standard (JIS) document. It is also in Ken Lunde's excellent book, Understanding Japanese Information Processing, in the section entitled "Japanese Hyphenation." While not all Japanese agree on the correct set of *kinsoku* characters, this set is a good default. Some applications allow the user to edit the *kinsoku* character set to their own liking.

Once you know that the current byte offset in your text is on or just before the graphic break, you need to see if that byte is part of a two-byte character. Then you need to see if the character is a character that can't end a line. Then you need to check the character after it to see if it is a character that can't begin a line. This can be repeated as necessary, for support of a string of *kinsoku* characters. For example, suppose the character on the graphic break (the break char) can end a line, but the character after it can't begin a line, causing the break char to wrap. However, the character before the break char is one that can't end a line, so you must then check the char before it, and so on, and so on, and... The example below is simplified to illustrate a particular case; actual code for an application would probably be organized differently.

```
UInt16 *      gStartLineKinsokuChars;      // chars that can't begin
                                           // a line.
UInt8         gNumStartKinsokuChars;       // number of chars above.
UInt16 *      gEndLineKinsokuChars; // chars that can't end a line
UInt8         gNumEndKinsokuChars;  // number of chars above.

// This function will return FALSE if the char at offset
// is not a valid break point. It checks the char after it,
// but not the char before it, for kinsoku.
static Boolean CheckGraphicBreak(UInt8 * textPtr,
                                           UInt16 offset,
                                           ScriptCode script);
{
       SInt16 result;


       // The textPtr starts at a known 'good' character
       // boundary. In this case it is the beginning of the
       // line, but it could be the beginning of the
       // stylerun.

       // Find out if script only has 1-byte chars. If so,
       // we assume it's ok to break at this char.
       if (GetScriptVariable(script, smScriptFlags) &
                                  (1 << smsfSingByte))
              return TRUE;

       result = CharacterByteType((Ptr)textPtr,
                                           offset,
                                           script);
       if (result == smSingleByte)
              return TRUE;  // In real life, you're not done
                                  // until you check the chars before
                                  // and after this one for kinsoku.
       if (result == smFirstByte)
              return FALSE;
       if (result == smLastByte)
       {
              UInt8         index;
              UInt16 theChar = *(UInt16 *)&textPtr[offset - 1];

              // Now we have a valid break on a 2-byte char.
              // We need to check if it's a kinsoku character.
              // This code checks Japanese kinsoku only, but
              // with a little work this could be extended to
              // all 2-byte scripts that don't break on spaces.
              if (script != smJapanese)
                     return TRUE;

              for (index = 0; index < gNumEndKinsokuChars; index++)
              {
                     if (theChar == gNumEndKinsokuChars[index])
                            return FALSE;
              }

              // Now we check the char after this one, in case it
              // is a char that can't start a line. First see if
              // it's a 1-byte char. In real life, there are 1-byte
```

```
                // kinsoku chars to check for.
                if (textPtr[offset + 1] == NULL ||
                        CharacterByteType((Ptr)&textPtr[offset + 1],
                                        0, script) == smSingleByte)
                        return TRUE;

                theChar = *(UInt16 *)&textPtr[offset + 1];

                for (index = 0; index < gNumStartKinsokuChars;
                                                        index++)
                {
                        if (theChar == gNumStartKinsokuChars[index])
                                return FALSE;
                }
        }
        return TRUE;
}
```

Listing 2: Checking Graphic Break Char with Kinsoku Processing

Hit Testing

Hit testing is another area in your text engine that demands the highest possible performance. When the user clicks in the text, any delay in setting the insertion point there will be noticed. Drag selection is another example of the same code working hard to find the character boundaries and setting the correct hilite area.

Some applications use a locally allocated cache of possible first byte character codes that they use to test a particular character in the text stream for "byteness" (that is, which byte of a possible byte-pair it is). This is simple to create, with the MacOS Toolbox call `FillParseTable()`. `FillParseTable()` returns in your pre-allocated 256 byte buffer all the bytes that can be a first byte of a two-byte character in the script you pass to it. Be aware that in some scripts, some character codes can be both the first byte of a two-byte character as well as the second byte of a two-byte character, depending on their context within the text stream. Therefore, you need more than just this information to successfully find out what kind of character the byte you're interested in is a part of. In a mixed stream of text with both one-byte and two-byte characters, using the parse table in a single pass over the text is much faster than calling `CharacterByteType()` for each byte. An example of this is below, in a sample function that goes through a text stream and counts the number of characters in it:

```
UInt32 CountCharsInScriptRun(UInt8 * textPtr, UInt32 length,
                                    ScriptCode script)
{
       UInt8         parseTable[256];
       UInt32 curByte, charCount;


       (void)FillParseTable(&parseTable, script);

       for (curByte = 0L, charCount = 0L; curByte < length;
                                                          curByte++)
       {
              if (parseTable[textPtr[curByte]] == 1)
                     continue;
              charCount++;
       }
       return (charCount);
}
```

Listing 3: Counting the Chars in Mixed-byte Text

Notice that because we started at a known 'good' boundary, we were able to test only the first bytes of the two-byte characters in the stream as we counted along. This code would not work in all cases if we started at an arbitrary point in unknown text, because of the ambiguity of the byteness of some character codes in some scripts. Caching the parse tables for all installed scripts in the user's system at launch time would further speed up your processing, so you wouldn't have to call `FillParseTable()` every time.

Measuring Two-byte Characters

On the Mac, all two-byte characters are the same width. In a future system software release, proportional two-byte characters will be supported, but up until now all two-byte-savvy applications assume mono-spaced two-byte characters, and even if proportional characters are supported, they will be mono-spaced by default so as not to break every application currently shipping.

Before the MacOS supported measuring two-byte characters with `TextWidth()`, a special code point in the single-byte 256 char width table was reserved for the two-byte character width for that font. In the Japanese and both Chinese scripts, this code point is `0x81`. In Korean script, it is `0x83`. This code point still works, even though Apple now recommends you use `TextWidth()` for all measuring of multi-byte or mixed text. In the future for proportional measuring, `TextWidth()` will probably be what you will use.

Below is an example of a function that measures any text, and returns the amount in a Fixed variable. This is useful if you are measuring text and the user has Fractional Glyph Widths turned on (meaning you made a call to `SetFractEnable()`).

```
#define JSCTCWidthChar       0x81
#define KWidthChar    0x83

typedef struct tagStyleRun {
       UInt32 styleStart;
       UInt16 font;
       UInt16 size;
       UInt8         face;
} StyleRun, *StyleRunPtr;

Fixed GetTextWidth(UInt8 * textPtr, UInt32 length,
                           StyleRunPtr styleRuns, UInt32 numStyles)
{
       ScriptCode            curScript;
       UInt32        byteNum, styleNum;
       Fixed                 totalWidth = 0L;
       ScriptCode            curScript;
       FMetricRec            curFontMetrics;
       WidthTable **  curWidthTable;
       UInt8                 parseTable[256];


       // loop thru each stylerun, measure its characters
       for (styleNum = 0L; styleNum < numStyles; styleNum++)
       {
              // Set up the port (in real life, you'd restore the
              // old settings when you exit)
              TextFont(styleRuns[styleNum].font);
              TextFace(styleRuns[styleNum].face);
              TextSize(styleRuns[styleNum].size);
              FontMetrics(&curFontMetrics);
              curWidthTable = curFontMetrics.wTabHandle;
              HLock((Handle)curWidthTable);
              curScript = FontScript();
              (void)FillParseTable(&parseTable, curScript);

              // loop thru each char in the stylerun
              for (byteNum = styleRuns[styleNum].styleStart;
                     (styleNum + 1 < numStyles &&
                     byteNum < styleRuns[styleNum+1].styleStart) ||
                     (styleNum + 1 >= numStyles && byteNum < length);
                     byteNum++)
              {
                     if (parseTable[textPtr[byteNum]] == 1)
                     {
                            if (curScript == smJapanese ||
                                   curScript == smTradChinese ||
                                   curScript == smSimpChinese)
                                   totWidth +=
                            (*curWidthTable->tabData)[JSCTCWidthChar];
                            else if (curScript == smKorean)
                                   totWidth +=
                            (*curWidthTable->tabData)[KWidthChar];
                            else
                                   totWidth += (Fixed)
                                   TextWidth(&textPtr[byteNum],
                                          0, 2) << 16;
```

```
                        byteNum++;
                }
                else
                        totWidth +=
                (*curWidthTable->tabData)[textPtr[byteNum]];
        }
        HUnlock((Handle)curWidthTable);
    }

    return (totWidth);
}
```

Listing 4: Measuring Mixed-byte Text

The above function still makes expensive calls like `FontMetrics()`, `FillParseTable()` and `TextWidth()` on each stylerun. It would be an even better idea to have a local cache of the width tables and parse tables of fonts you know are in the document, so you don't have to rebuild them every time the user clicks or drags or types in the text.

So, now that you have a relatively fast way of measuring the text, you can use it to find the pixel value of any character in the text, and use that for your internal CharToPixel and PixelToChar logic. Or, you can use the MacOS Toolbox calls `CharToPixel()` and `PixelToChar()`, which will always work on any script but may be slower.

Localizing Your Application for Japan

Now that we have reviewed a few of the basic text engine issues for handling two-byte text, there are a few things about Japan in particular that make localization a challenge.

Japan is possibly the most interesting major software market to localize for if you are interested in text and text layout. It is a mature market, with a diverse number of products enjoying many millions of dollars in sales each year. The Macintosh has a larger market share there than in the U.S. or Europe. Text in Japan has traditionally been difficult to input and output using machines, and the use of text in graphic design requires that the text layout be extremely flexible. The characters are complex (so complex that bolding them may make them illegible), and emphasis or adornment has forms that use background shading, different types of lines around the text, and even dots or ticks above or to one side of each character. Condensed and extended faces have different results on PostScript® printers than they do on QuickDraw displays. Bold and italic faces were not supported on the first PostScript® Japanese printers. Underlines are not drawn by QuickDraw when the font is a Japanese font. These last two things might be fixed in future releases of the system software, but for now the application developer must work around them.

For underline, you must draw a line under the text. The reason QuickDraw doesn't draw it for you is that it usually uses the font's baseline as the underline location, but Japanese fonts' two-byte glyphs take up more room and descend below the baseline. Where you draw your underline is up to you, but take a look at how other Japanese programs do it and make it fairly consistent.

Vertical text is pretty much a checkbox item nowadays in Japanese word-processing programs. Most novels and many magazines are layed out vertically, but until recently computers were horizontal-only. While the Windows95® APIs support drawing text vertically, the MacOS still does not, outside of using QuickDrawGX typography (which is excellent, by the way). In

comparing vertical text to horizontal text, several things change about the line layout: The first line starts at the top right, and the text flows down to line-end, then wraps to the next line, which is to the left of the first line; the baseline is generally considered to be in the center of the line; underlines are drawn to the right of the text, as are emphasis dots; two-byte characters are not rotated, but single-byte characters are, 90° clockwise; certain characters have vertical text variants, like many punctuation characters. Where these variants are in the font can be found in the `'tate'` table in the font ("tahteh" means "vertical" in Japanese).

Rubi are small annotation characters, placed above, below or to the side of the text they annotate. Usually they provide pronunciation guidance for unusual or hard-to-pronounce Kanji characters.

Date formats in Japan include the current year of the emperor's reign; again, supported on Windows95® but not on MacOS. It is up to the application to support these formats if so desired. Also, date formats 2 and 3 produce identical results, due to the fact that the abbreviated month and the long month are the same thing in Japanese. Japanese applications may opt to substitute a different format in one of those formats' place.

Find and Replace needs to be expanded to include the different types of characters used in Japanese. Standard Japanese text may contain any of the following types of characters: one-byte Roman, two-byte Roman, one-byte numerals and symbols, two-byte numerals and symbols, one-byte katakana syllables, two-byte katakana syllables, two-byte hiragana syllables, and two-byte Kanji characters. The hiragana and katakana characters are equivalent in terms of the sounds they represent in Japanese, so a good Find/Replace function should include an option to find the search string in either syllabary.

Sorting in Japanese is difficult because the Kanji characters can have different pronunciations depending on their context. To sort Kanji correctly, you need a separate kana key field that indicates the pronunciation and you sort on that. Also, MacOS `CompareText()` doesn't sort the long sound symbol correctly (that symbol changes sound depending on the character before it, but MacOS always sorts it in the symbols area), so for linguistically correct sorting you need to write your own sorting routine.

If your application supports character tracking using the Color QuickDraw function `CharExtra()`, be aware that the `CGrafPort` member `chExtra` only uses 4 bits for signed integer values and the other 12 bits for the fraction. The value you pass to `CharExtra()` is a Fixed value of how many pixels you wish to track out (or in) the text, and QuickDraw divides that by the current text size, to arrive at the `chExtra` value. This means that if the tracking value you pass to `CharExtra()` is greater than 8 times the text size, the `chExtra` field will go negative, and your text will be drawn incorrectly. Unfortunately, Japanese text is routinely tracked out beyond this limit in many applications. The only workaround is for you to draw the text one character at a time, and use the QuickDraw pen movement calls like `MoveTo()` to move the pen yourself. The same is true for `SpaceExtra()`.

Inline Input

Inline input of Japanese, Chinese or Korean is a way of using an intermediate program (called an Input Method) to translate your keystrokes into the many thousands of possible characters in those languages, all in the same place on screen that you would normally see characters typed in the line. In Japanese, the Input Method changes your keystrokes into phonetic Japanese kana

characters, then converts some of those characters into Kanji characters to form a mixed kana and Kanji sentence. Then the user hits the return key to confirm the text in the line, ending the inline input session. Inline input on the Mac on System 7.1 or later uses the Text Services Manager (TSM). If your application uses TextEdit as its main text engine, you can support inline input quite easily using TSMTE. If you have your own text engine, you will need to do more work to support TSM Inline Input.

TSM uses AppleEvents to send and receive data between your application and the Input Method. You must implement several AppleEvent handlers, the most complex of which is the `kUpdateActiveInputArea`. In that handler, you must draw the text in all its intermediate stages, as the user is composing and editing the Japanese sentence before s/he confirms it to the document. If there is text after the so-called 'inline hole,' you must actively reflow the text if such editing causes the length to change. Each time the user makes a change, the text in the inline hole is received from the Input Method in an AppleEvent. The application draws it in the text stream, along with special Inline styles that help the user tell which text in the inline hole is raw (unconverted) text, which is converted text, which is the active phrase, where the phrase boundaries are in the inline hole, and other information.

After implementing the TSM support in your application, it is imperative that you test it with third-party Input Methods. At the time TSM was introduced, the documentation for how to write an Input Method was still a little spotty. This resulted in each Input Method handling text slightly differently. Also, Kotoeri, Apple's Input Method, has fewer features than the leading third-party Input Methods. Be sure to test your application with all of them you can find, so you can verify that it won't crash or produce strange results. Some Input Methods have strange quirks, like always eating `mouseDown` events, or having different requirements about how large a buffer they can handle without crashing. This knowledge comes from testing, and sometimes can be found on the Internet in Usenet newsgroups (in Japanese).

What About Unicode?

Unicode is being billed as the latest panacea for the problems of internationalization. What does Unicode give you? Where does it fall short?

Unicode was designed to solve one problem: There are many incompatible, overlapping encoding schemes for different languages, and supporting all of these encodings is a complex problem. What if there was a single encoding scheme that supported all the writing systems of the world, and guaranteed that you could display text in all the languages Unicode supports if only you had the right Unicode font for each language? Unicode tries to be that encoding.

For Japanese text data, the MacOS and Windows95® use Shift-JIS internally, while Rhapsody and WindowsNT® use Unicode. On the internet, most Japanese text is encoded using the 7-bit ISO-2022-JP standard. Whether or not you use Unicode to represent text internally to your application, you will have to support all three standards for full file and data compatibility with the rest of the world. In Unicode, all characters are two bytes long. So, you no longer have to worry about testing for byteness in a Unicode stream. However, all ASCII characters are represented with a leading `0x00` in Unicode. So you can't have loops that look for a terminating `NULL` in a C-string. And, all your formerly one-byte text doubles in size unless you explicitly compress it (and then you lose the byteness testing advantage).

Whether or not you think testing byteness is too complex or expensive to do, you should know that Unicode also does another controversial thing: For the so-called "Han" languages (Japanese, Chinese, Korean) that use characters that originated in China, it attempts to unify them into one codepoint for each character judged by the Unicode Consortium to be unique, even if it has variant forms in each language. The same is true for Arabic languages (Persian, Farsi). Because of this, you cannot tell what language a character is in just from its codepoint. Unicode was not designed to be a multi-lingual solution, in that representations of Chinese and Japanese in the same document will have overlapping character codes, requiring the OS to provide a parallel linguistically-coded data structure to render the glyph forms appropriately to each language. This might be another version of today's font/script/language relationship on MacOS. As you can imagine, the Chinese, Japanese and Korean governments have each published competing encoding standards to Unicode, labeling the latter as something designed by foreigners who didn't understand the issues (both political and linguistic) involved in trying to make a worldwide encoding system.

Another issue about Unicode is that although it can represent 65,536 characters, there is not enough space for all the Han characters and their variants, plus all the other languages that Unicode currently supports. New languages are becoming computerized as more countries join the Digital Revolution and the Unicode Consortium cannot give space to all of them. Preferring the flat encoding model, they came up with another standard that uses four bytes per character (the ISO 10646 encoding standard). Given that on the Internet, where many languages need simultaneous support on computers, bandwidth is at a premium, I would prefer using the ISO 2022 standard of mixed-byte (7-bit and 14-bit characters) plus the escape codes that tell you what language the current stream is in to sending 32-bit characters through the wire. Since most web pages use this encoding, expect your OS to provide utilities for encoding conversion (like the MacOS Encoding Converter debuting soon on a Mac near you).

Cross-Platform Development Issues

Going cross-platform is already complicated without having to think about internationalization. Should you have separate codebases for maximum use of each platform's unique features? Or should you have a single codebase and use an emulation layer for the other OS's APIs? Each has its advantages, but for this paper I can speak to those of you who have a joint codebase, and tell you about some of the things that the Windows platform lacks that you have to write yourself for multi-byte support and internationalization.

Windows has no Script Manager. There is no Gestalt Manager. It cannot support multiple two-byte codepages at the same time. It uses totally separate fonts for vertical and horizontal text. It supports proportional kana in Japanese, so you can't assume all two-byte characters are the same width.

If your code uses the Script Manager routines heavily, then you will have to write them yourself on the Windows side. All the convenience of the MacOS's international routines comes very clear when you try the same things on a PC!

Also, Japan once again has its own special challenges. Until Windows came out in Japan, each computer manufacturer made its own proprietary OS and hardware. Even floppy disks were incompatible with each other. Now, most companies have adopted the Intel PC standard, but NEC continues to manufacture its own line of incompatible PCs. NEC has such a huge share of the market in Japan that it has teamed up with Microsoft to produce its own version of Windows95® for NEC. So when you buy Windows95® in Japan, you find there are three

versions: MS Windows95 for Intel, MS Windows95 for NEC, and NEC Windows95 for NEC. All three versions are basically the same feature-for-feature, but the drivers are different and you need to test your application on each platform to verify compatibility.

On the hardware side, you will find that Japanese hardware is different: They use different displays, different keyboards, different printers, and different floppy formats. The drive lettering on NEC machines is different from Intel PCs: The hard disk drive is labeled 'A:' on one and 'C:' on the other. Make sure your installer isn't hard-coded to install on drive C:.

Conclusion

As we have seen, internationalization of your software on MacOS is not very difficult to do, and it is to your benefit to try and enable as many users as possible to enter text in their own language when using your program. We have also examined Japanese localization in more depth, and demonstrated that Japanese language applications usually require some amount of new features designed specifically for that language's needs and conventions. As more markets around the world reach maturity, you can be sure that there will be ample opportunity to differentiate your product by adding locale-specific features. It is these locale-specific features that will tell your users that they are valued customers, and that their needs are being addressed in a very specific way. For your product, especially if you are in the initial designing phases, I would recommend you try to make it as easily expandable as possible. Design generic internationization into the core modules, while leaving open the opportunity to add locale-specific features for certain markets like Japan, as you see your product's market expand and rise in success.

Bibliography and Related Reading

Apple Computer, Inc. Inside Macintosh: Text, Menlo Park, CA: Addison Wesley, March 1993.

Apple Computer, Inc. "Technote OV 20, Internationalization Checklist," Cupertino, CA: Apple Computer, Inc, November 1993.

Griffith, Tague. "Gearing Up for Asia With the Text Services Manager and TSMTE," Develop Issue 29. Cupertino, CA: Apple Computer, Inc, March 1997.

Apple Computer, Inc. "Technote TE 531, Text Services Manager Q&As," Cupertino, CA: Apple Computer, Inc, May 1993.

Lunde, Ken. Understanding Japanese Information Processing, Sebastopol, CA: O'Reilly & Associates, September, 1993.

See also Ken Lunde's home page at `http://jasper.ora.com/lunde/`. More information about multi-byte text processing on computers.

About the Author

Nat McCully has been at Claris in the Japanese Development Group for the last 6 years. He has worked on numerous Japanese products, including MacWrite II–J, Filemaker Pro–J, Claris Impact–J, ClarisDraw–J, and ClarisWorks–J. He speaks, reads and writes Japanese, and enjoys traveling in Japan. He is currently working as Technical Lead on the next release of ClarisWorks–J.

On Having Two (or More) Heads
Real-Time Programming in Spite of the MacOS

© 1997 by Rainer Brockerhoff
rainer@machome.com.br
http://www.machome.com.br/delta/

Abstract

Using the MacOS for acquiring and displaying real-time data is a problem with not-too-well documented solutions. Recent advances like the Thread Manager and (for Internet applications) OpenTransport, as well as asynchronous I/O, are of course helpful and indeed indispensable for whoever tries to do real-time programming but it is not always clear how to deploy these facilities to the greatest advantage. The MacOS, because of its single-user, single-application origins, still places some restrictions on what can be done in this regard. Some of those restrictions can be overcome; this is what this paper is about. Reasonable familiarity with Macintosh programming and PowerPlant is assumed.

Introduction

For some years my company, BESE Bio Engenharia of Belo Horizonte, Brazil, has been manufacturing cardiac monitors for Intensive Care Units, based on the Motorola 68000 processor. In late 1996 we designed a new product : Central BESE (which I'll call **CB** hereafter, for brevity's sake), a central monitoring station based on the PowerMac platform which acquires real-time data sent out by up to 12 individual cardiac monitors and displays all data on the Mac's screen. The acquired data are also continually saved to disk for documentation purposes. As you can imagine, such a record must be complete and uninterrupted.

I have been successful in working around most of the restrictions of the MacOS for our application. Since many of those restrictions and solutions are not specific to our application, I've built an example program as an illustration for this paper. The program is built in CodeWarrior 12 (or Pro 1, as it has been renamed) C++ with PowerPlant, as is our application. For performance considerations, **CB** only runs on the PowerPC; there is no 68K version.

The example program included with this paper, as well as the relevant classes, is called "Hydra", after the many-headed monster of Greek mythology. Recall that the mythological Hydra grew two more heads for every one that was cut off! Fortunately, such recursive growth didn't happen in our case. Hydra also is a PowerPC-only program to avoid cluttering up the example code. However, all solutions presented here are workable on the 68K too. Porting Hydra to the 68K environment is left as an exercise for the student [:-)].

I'm assuming that you have some experience with coding applications using PowerPlant in the CodeWarrior environment. But even if you're just beginning to program on the Macintosh you will hopefully get some insight into what is going on.

The Problem

Suppose you have several sources sending in data to your application. They might be serial ports, as in **CB**, our original case, or data streams coming in over the Internet, or they might be simply simulation processes, as in the Hydra Example application. If you're building a game, you might have only low-volume data input from a joystick, but you will have several simulation processes which depend on those data. You need to read in each data stream, massage the data, and continually display a suitable view in a window (or part of a window). At the same time, the user may be asking you to do other things, like changing the display format, setting and saving preferences, printing out accumulated data, and so forth or even switching you to the background to edit a Word document or run Netscape in the foreground. You're not allowed to lose data or stop the display for more than a fraction of a second. What now?

We'll discard preemptive threads since they don't work on the PowerPC (and even on the 68K, they have serious restrictions on Toolbox calls, which we need for data display). Multiprocessing is not considered for much the same reasons. By all accounts, Rhapsody's upcoming "Yellow Box" will make our task much easier, but "Blue Box" applications will still be able to use the solutions detailed in this paper.

A Bit of History

The MacOS uses cooperative multitasking; that is, well-behaved applications periodically yield CPU time to each other by calling `WaitNextEvent()`. In 1984, when the first Macintosh came out, it was impossible to run more than one application at the same time. Even so, Desk Accessories (DA's), which in those days were very restricted mini-applications, could be run on top of the main application. One of those DA's was the famous "Alarm Clock", which of course updated itself every second and checked if its alarm had gone off.

Probably because of that single DA, System 1.0 already had a `SystemTask()` trap which was designed to periodically yield time to DA's. Well-behaved applications were enjoined to call `SystemTask()` "at least 60 times a second", to quote "Inside Macintosh". No thought was given to preemptive multitasking, which would have implied using more complicated (and therefore slower and code-bloating) techniques for nearly all components of the system software. Since CPU time and memory space were at a premium, the designers had no other choice.

Preemptive multitasking might have been introduced later on, when MultiFinder was introduced. MultiFinder, which was later to become integrated into the System, fooled applications into thinking they were alone on the machine. First, `WaitNextEvent()` was introduced to combine the methods of the old `GetNextEvent()` and `SystemTask()` traps. Then, every time control was yielded to another application, parts of low memory (including trap dispatch tables) were switched for the new applications's .

This was a way to (1) stay compatible with older programs, (2) allow new programs to become MultiFinder-compatible with very little change, (3) most importantly, allow MultiFinder to run on 68000 Macs. Later chips had memory-mapping hardware either installed alongside the CPU (68020) or built-in (68030 and 68040); had they restricted MultiFinder to those machines, this might have allowed the System to go to a preemptive model with multiple address spaces… now finally to be delivered in Rhapsody.

However, hindsight is always 100% accurate and the installed base of 68000 machines probably was considered too large to be ignored. Let's now try to make our real-time application appear to work — at least from the user's standpoint — as if the Macintosh had gone the preemptive way instead.

Running in the Background

This problem is the worst. With cooperative multitasking you can't be assured of getting enough time if you're in the background, since you depend on the foreground application calling `WaitNextEvent()` often enough or with a long enough wait time. If you have others in the background they may not yield enought time to you, either.

The solution, unfortunately, is of the cut-off-the-head-to-cure-the-headache type. In **CB**, we kill all other applications, including the Finder. Basically, **CB** scans all active processes using `GetNextProcess()`, identifies the Finder and all applications (excepting itself), kills all of them by sending them a "Quit Application" AppleEvent, and lastly, kills the Finder.

Getting Several Data Streams at Once

There are three main approaches to the problem of simultaneous data streams.

The first approach shows up in some games, which use asynchronous `PBReadAsync()`'s to acquire data. Every completion routine immediately starts off another `PBReadAsync()`, pushes the data into a first-in first-out (FIFO) queue and posts a custom event to the OS Event Queue. You can't allocate memory or display something in the completion routine, so you have to postpone processing to the main event loop. The main event loop eventually gets the event (as well as user interface events) and pulls the data off the FIFO to process it.

There are some possible problems with this approach. All your data-handling is serialized; you process one block of data at a time. In effect, you are multiplexing your several data streams into a single stream, and just adding some info to each block or packet, so that you can tell which stream this information comes from. This works if you don't have to do much postprocessing for display, or if you have to consolidate those streams anyway before displaying them. However you may have to demultiplex your data stream again, if (for instance) each one is displayed in a different window; you end up having things like arrays of pointers to windows, and so forth, which are cumbersome to manage. If your display routines are complex or time-consuming, you can't have several windows updating at the same time because your application is serialized.

The second approach is the one we've used in **CB**, and which is illustrated in Hydra. You use the Thread Manager to start up a thread for each data stream. Every thread handles and displays its own data. You use the main thread to do user interface (UI) processing.

There are some things to be aware of with respect to this approach – they will be explained in due course.

The third approach is a hybrid one commonly used by Internet clients (or servers), which take advantage of OpenTransport's facilities to postprocess data coming in over OT endpoints. Since those facilities use the Delayed Task Manager, you have the same limitations as the completion routines used in the first approach. However, here you use FIFO queues to pass the

data off to threads (instead of to the main event loop). We won't discuss this approach further, both because I've never used it myself [;-)] and because once you get data into the threads, it reduces to the second approach.

Blocked by a Mere Mouse

Suppose you've adopted any of those approaches; you'll still find everything grinding to a halt while UI action is going on. While the user pulls down a menu or drags something with the mouse, all other threads stop. Worse, many UI actions assume that no other activities are going on; menus, for instance, save and restore whatever is below them and are absolutely sure the screen doesn't change while they're pulled down. Of course, asynchronous I/O is not blocked by the mouse… but eventually you'll be running out of buffers to store data, and you'll need too much time to catch up, once the user releases the mouse!

Here too, there are several possible solutions. There are two different issues to consider here :

- You will need to give time to your threads during the UI interaction, and in consequence,
- You will want to let the threads display data without messing up the UI display.

If you have used PowerPlant's application/commander classes, you know that they assume that updating is done by just calling `Refresh()` when you change a Pane's contents. Then the main event loop, which is hidden from you, gets an update event and redraws the changed pane. However, this usually introduces an undesired delay into the display process. In **CB**, for instance, we have to display physiological graphs (like electrocardiograms) in as close to real-time as possible; drawing a new chunk a few times every second is not acceptable. So you need to update the real-time display without using update events, but still handle update events if they're received — when a window is uncovered, for instance.

Finally, contrary to what happens in single-threaded applications, you need more cooperation from the main event loop, which usually runs too often. Your data-handling threads need all the priority they can get, and they will reluctantly allow the main thread to handle events and UI interaction whenever absolutely necessary, but not as much as usual. For this you need to put some additional intelligence on top of PowerPlant's thread classes.

As it happens, taking care of all of these issues has the happy side effect of also allowing you to interleave display of your data without having all those windows (or panes) interfering with each other.

The Example Application and the Hydra Classes

Let's look at the example application to illustrate our points so far. The `Hydra.cp` and `Hydra.h` files contain our special classes which hide nearly all of the details from the main application. In the first place, there are `HydraApplication` and `HydraWindow` classes. They are subclasses of `LApplication` and `LWindow` respectively, but all their special functionality is user-transparent; there are no additional methods available. You have to be careful when overriding their internal methods, of course; normally there will be no need to do so.

`HydraApplication` calls the necessary initialization (and, later on, termination) methods for Hydra, besides overriding some base methods to get the needed thread behavior. It also sets up your main thread. `HydraWindow` takes care of menu shielding for your real-time windows, although you also have to do one additional call for each real-time pane.

The Hydra class contains all static routines and variables we need, most of them accessible only by the other Hydra classes; it also serves as a stack-based state saver, as we'll see later. See the Hydra.cp file for detailed explanations of the available methods; I suggest you at least page through it before reading any further.

All this means that if you open up HydraExample.h you will see that the only noticeable deviations from a normal PowerPlant program are that we subclass HydraApplication and HydraWindow instead of LApplication and LWindow.

In the same way you'll find little that's new in the HydraExample.cp file. This example application reproduces, in its own way, one of my first Apple II programs, from way-back-when in 1977. The ExampleWindow class simulates two different types of real-time process; one that runs at full speed, depending only on the input data rate, and one that repeats at a set rate. Since we don't have a ready-made data source easily available, the first type is simulated here by drawing colored lines as fast as possible with occasional waits, and the second type by drawing colored rectangles every so often. To simplify things we don't use any panes inside our windows, but a practical example would of course draw inside some LPane subclass.

Be cautioned that, this being just an example application, very little error-checking is done; the Hydra classes themselves should at least throw some exceptions to be usable in a real-world application.

Each ExampleWindow has its own LSimpleThread process to do its processing. Here this is simply an infinite loop calling DoUpdate(), which updates the window according to its type. The type is stored in the window's pane ID and may be equal to lines, in which case DrawLines() is called repeatedly, or rects, which causes DrawRects() to be called instead.

In practice the window's process would start up asynchronous PBReadAsync()'s to get data, staying suspended until data are actually available, and then displaying the results in its parent window. Meanwhile, another PBReadAsync() would be chained and waiting for more data. I didn't include support methods for asynchronous resumes into the Hydra classes, but you can use the standard LThread methods.

As you can see, having one thread for each data source embedded in each window (or pane) makes housekeeping very easy since you don't need to worry about matching up your windows, threads and data sources.

Now let's see how we keep the thread's drawing from interfering with normal window updating, with each other, and with menus.

Normal and Fast Window Updating

Normally in PowerPlant every time you wish to update a pane you call the pane's Refresh() method. This marks the pane's rectangle as needing an update and generates an update event. As soon as the main event loop gets the update event, it passes the event down the hierarchy of views and panes until eventually the desired pane's DrawSelf()is called. At this time the

window's GrafPort is all set up and DrawSelf() can just call the necessary QuickDraw methods.

PowerPlant's pane, window and view classes take care of setting up the correct GrafPort by calling FocusDraw() (or one of it's variations). LView caches the current GrafPort to avoid some overhead and does not change the GrafPort if it's already selected. This speeds up cases like when you're redrawing a window containing several panes, for instance. And of course all this happens as well when you uncover a window, since parts of it will need to be updated. In this case the necessary update event is generated by the system.

We can't use the update event mechanism for real-time display as it is too slow therefore the thread has to draw directly to its parent window. It does this by calling FocusDraw() followed by Hydra::ShieldMenu(mSuperView) and then doing the necessary drawing (ShieldMenu is explained below in the menu section). But of course you also have to redraw some or all of the window when an update occurs. The easiest way is to use a LGWorld to hold an image of the window (or pane) and have DrawSelf() simply copy this image to the screen. From the thread you can either draw directly into the LGWorld and then call DrawSelf() (possibly clipping to just the updated part) or do as I did in the example, drawing the same thing both to the screen and to the LGWorld .

In your thread, if you simply call FocusDraw() and Hydra::ShieldMenu() to draw something, and then yield to other threads, you won't have any trouble. But what if your display updating is complex and you have to yield the CPU to other threads inside DrawSelf()? If the other threads also draw something (or change the current GrafPort), this may cause PowerPlant's caching mechanism to fail.

You can of course walk through your program and call LView::OutOfFocus(NULL) every time you change the GrafPort, or after you've finished with a drawing section. When using the Hydra classes there is a more convenient way to save and restore the GrafPort when needed :

```
MyDrawingFcn() {
    Hydra saver;            // this stores the current GrafPort
    ...                     // any convenient name will work
    ::SetPort(otherPort);   // (probably non-PowerPlant GrafPort)
    ::EraseRect(&someRect); // draws something
};                          // Hydra's destructor restores the
                            // GrafPort and clears the view cache
                            // if necessary
```

Other parts of the Hydra classes use this technique internally —when doing a thread switch, for instance. This means simple threads never have to worry about GrafPort's between switches; indeed, this technique was not needed for our example program.

Handling Menus

Let's have another look at one the drawing routines called by the example's thread. Both DrawLines() and DrawRects() look like this:

```
 {
    Hydra::ShieldMenu(this);  // shields menus
    ...                       // draws to the screen
    mGWorld->BeginDrawing();
    ...                       // draws to the LGWorld
    mGWorld->EndDrawing();
    Hydra::Sleep(oneSecond/8);  // yields to other tasks
};
```

Why, you might ask after reading the preceding section, is it necessary to call `FocusDraw()` and `Hydra::ShieldMenu()` every time before drawing? You might think that both `Hydra::Sleep()` or `Hydra::Yield()` preserve the `GrafPort` correctly, and so `FocusDraw()` would need to be called only once. That's indeed true, but this doesn't guarantee your menus are protected.

Indeed the `GrafPort` is preserved, but its clipping region may be outdated, because of the fact that a menu might have been pulled down (or snapped back up) while the thread was not active. The `Hydra` class implements its own `LMenuBar` subclass and `MBarHook` procedure to detect when a menu is pulled down and calculates the screen region hidden by the menu. Then, when the menu snaps back up, all windows redrawn while the menu was down are updated and the menu region is cleared. So , every time you want to draw to a `HydraWindow` or one of its panes, the saved menu region (if non-empty) has to be punched out from the window's clipping region. This enables us to draw into the window even while a menu is pulled down.

The way to do this is to call `Hydra::ShieldMenu()` before drawing (supposing you're already focused). The parameter to this routine must be either `mSuperView` (if you're in a pane) or `this` (if you're in a view or window). `HydraWindow::FocusDraw()` is already overridden to do this, but you have to do it separately for each real-time pane; either by overriding the pane's FocusDraw() or by explicitly calling `Hydra::ShieldMenu()` inside its `DrawSelf()` method.

Now if `Hydra::Yield()` is called it may yield control to the application's main thread. As well see later on, only the main thread checks the event queue and therefore the user interface; if the main thread gets a click in the menu bar, it pulls the correct menu down. However, as we'll see later, Hydra has the Menu Manager's loop call `Hydra::Yield()`! This means that, unlike with conventional applications, your threads continue running while menus are pulled down.

The downside of this is of course that, once `Hydra::Sleep()` or `Hydra::Yield()` have been called, you can't be sure that the menu status is still the same and your window's clipping region may be invalid. So be sure and call `Hydra::ShieldMenu()` again before drawing after yielding control; you don't need to refocus because the `GrafPort` is saved.

An unfortunate limitation of the Hydra classes as they stand are pop-up menus. For some weird reason the system 's standard `MDEF` does not call the `MBarHook` procedure when opening a pop-up (or lower-order hierarchical) menu. Therefore you have to be careful never to open such a menu overlapping a `HydraWindow`. The only full solution is to install a custom `MDEF` for pop-up menus; this is left as an exercise for the reader (meaning I haven't had time for it, myself).

There also is a weird bug in the standard `MBDF` procedure. If you pull down a menu and then drag to an empty region of the menu bar, the menu snaps up immediately but the application is not notified; try doing that in the example application and a "ghost" of the last menu will stay punched out from the underlying windows. The correct thing would be for the `MBDF` procedure to call the `MBarHook` method with an empty rectangle, to tell us that no menu is currently pulled down. Perhaps some volunteer will write a corrected custom `MBDF` (hint, hint)?

Handling the User Interface

PowerPlant suggests that events be handled inside the main thread only. This is excellent advice the Thread Manager peers ahead into the event queue and gives scheduling priority to the main thread if an OS Event is in the queue. If you have other threads handling events, this can cause misscheduling, with the result that strange things may happen; mouse clicks may appear to get lost, for instance.

`Hydra` also looks ahead into the event queue to let the main thread run. As a general rule, you should not use idlers and repeaters in a threaded application; if you need things to happen periodically, create a thread to do that instead. Therefore the main thread can sit around idle most of the time and should give priority to the other threads. I implemented special `Hydra::Sleep()` and `Hydra::Yield()` methods to take care of this; they leave the main thread suspended and wake it up either for processing a queued event or after a certain time has passed. This time is set by calling `SetWNEFrequency()` with the desired frequency; 10 Hz is the default.

If you look at the code carefully, you will notice that the main thread just seems to loop calling `Hydra::Yield()` instead of `::WaitNextEvent()`. Actually `Hydra::Yield()` checks if it's being called by the main thread; if true, it does the event handling and sets the necessary timers to wake up the main thread again if an event is queued or the set time has passed.

A side-effect of this technique is that certain things are tied to the frequency set by `SetWNEFrequency()`. This includes calling repeaters and idlers (if any), updating menus, setting the cursor and handling help balloons. So don't set the frequency too low (and have your menus seem too sluggish) or too high (and lose time for your other threads). The default, 10 Hz, seems a reasonable compromise.

Of course `Hydra::Yield()` has to be called inside every UI loop that executes outside the main event loop. In **CB** we installed callback procedures for menus, dialog boxes, and so forth, but the Hydra classes simply patch `::WaitMouseUp()` and `StillDown()` to call `Hydra::Yield()` after their normal methods. `WaitMouseUp()` calls `StillDown()` internally, so we need to be careful not to yield twice if `WaitMouseUp()` is called.

If you look at the code you'll see that both patches are "head" patches, since they call the original trap first and then execute something else. This is usually considered unsafe on the 68K but there's no such problem with the PowerPC. However, in this case, a head patch is absolutely necessary; since both traps look ahead into the event queue, and `Hydra::Yield()` also does, a tail patch won't work correctly.

Dragging Windows and Other Special Cases

`HydraWindow` handles window dragging with no problems. As we'll see later on, `Hydra::Yield()` is implicitly called while dragging the window, so our threads don't stop while a window is dragged. This would introduce complications since the window outline (produced by `::DragGrayRgn()`) would have to be adequately shielded from `HydraWindow` updates. Fortunately this can be avoided by dragging the whole window around, instead of just an outline; this is a little slower but looks great. The same goes for resizing, of course.

The `::ZoomRects()` and `::ZoomRegions()` procedure are not usable, both because the zooming outlines are immediately erased if they go over `HydraWindow`'s, and because you don't get control back while zooming is going on. In **CB** we coded our own procedures that call `Hydra::Sleep()` between the zoom steps and temporarily adds the zooming regions to the menu region. I didn't have time to include this into the Hydra classes, however.

If you use PowerPlant's drag-and-drop classes to drag icons and other things around, there are two problems; the easy one is transparently handled by `Hydra`. Try dragging a window picture to the desktop and a picture clipping will appear. You'll notice that window updating continues while you're dragging, and the drag outline is not overdrawn because `Hydra` installs a custom tracking procedure to take care of it. This would not be necessary if the Drag Manager called `::StillDown()`, like everybody else, instead of `::Button()`; oh well.

But try dragging the mouse entirely out of the application's windows and updating will stop. This is due to the way the Drag Manager works; if you drag onto another application's windows you get switched out and your drag callback doesn't get called anymore. There are three ways around that : (1) kill all other applications, as we've done in **CB**; (2) cover all the screen with your windows; (3) pester Apple to fix the Drag Manager to call `::StillDown()` instead of`::Button()` after first switching your application back in.

If you use LDialogBox or other PowerPlant window classes that are handled and updated by the main thread, you need not subclass `HydraWindow`, as the usual update mechanism works OK — unless you plan on dragging or resizing them, of course. No menu shielding is necessary, either. In the example application, the "About" box is handled that way. Needless to say, you never should call `::Alert()` and its derivatives, since those have their own internal event loops and you won't get control back until the alert is closed.

Help Balloons are a special case. If you set your help resources to draw help balloons as windows, instead of saving the screen underneath, as menus do, the balloon's window automatically shields your `HydraWindow`'s — although, in the current implementation, the update region isn't always generated correctly, and you may sometimes see balloon-shaped holes in your windows; sometimes just the balloon's tip is not updated. Unfortunately there seems to be no way around this bug.

The main Help Balloon problem lies with balloons drawn by `::HMShowMenuBalloon()`. This procedure never generates balloon windows and therefore those balloons will be overdrawn if they overlap `HydraWindow`'s . Unfortunately the standard `MDEF` procedure calls `::HMShowMenuBalloon()`, so you'll have the strange situation that balloons for menu titles work correctly, but balloons for menu items don't. Again, coding a custom `MDEF` (or patching `::HMShowMenuBalloon()`) would solve this problem.

Floppy disks and LocalTalk networks both have the nasty habit of disabling interrupts and hogging the CPU much more than you would think. Avoid them like the plague! Ethernet will probably work OK, although I've done no real-world testing.

Printing — the Final Frontier

Sadly, printing while acquiring real-time data is still a serious, unsolved problem. Once you get into the printing routines they assume full control and you can't do much to keep your threads going. Large read buffers are a must but even so I found that printing to an ink jet often tied up everything for over a minute.

This should theoretically be somewhat eased by calling `Hydra::Yield()` inside your print item hook procedure (pointed at by `pItemProc` in your `TPPrDlgRef`), but usually printers just don't call this procedure often enough to make any practical difference. I tried it with an HP ink jet, and the procedure was never called during imaging and only every couple of seconds during data transmission. And if you have background printing turned on, you also get the extra performance hit from the extra background task.

For **CB** we had to resort to the somewhat desperate measure of writing our own printer driver. Since we, fortunately, needed only to dump single graphics pages to a bundled ink jet printer, this wasn't too great a hardship — we simply used the common PCL-3 printer language, and that's general enough for our needs. If you absolutely need to print from your application, this may be only way for you to manage it. Patching into the serial drivers to call `Hydra::Yield()` periodically might work too; we plan to investigate this possibility later on.

Final Words

As you saw, it is possible to display real-time data in a threaded application if you are aware of the problem areas. I hope this paper showed you a viable solution to the problems discussed. Study the enclosed source code for more details and please send me your comments, critiques and —hopefully — better or more complete solutions.

You may use the Hydra classes in your application at no cost; however, the application's "About" box should clearly state "Hydra classes are Copyright ©1997 by Rainer Brockerhoff". I'd also appreciate getting a demo copy of your application. Be warned, however, that those classes were pulled together from all over **CB** for the example application and were not submitted to industrial-strength testing. All such use is at your own risk.

Many thanks to Dave Johnson of Apple Computer for alerting me to MacHack, to Christopher Haupt, Bill Worzel and the other folks at MacHack for the opportunity to publish this paper, and to Háj Ross for reviewing an early version and helping me with some of the fine points of technical English. Any remaining errors and inaccuracies are, of course, my own. An indispensable tool for debugging all this was Alessandro Levi Montalcini's "MenuBall" control panel; grazie tante, Alessandro! Every serious real-time programmer should get a copy.

Implementing Threaded IO on the Mac OS

© 1997 by Jonathan "Wolfie" Rentzsch
jonathan@u-s-x.com

Abstract

This paper explains input/output (IO) on the Mac OS. After detailing the two IO models, the paper provides an explanation of how two match the Thread Manager with Mac OS IO with three examples. This paper finally introduces a new method along with the code behind it.

Introduction

Casey would waltz with the strawberry blonde
And the band played on
He'd glide 'cross the floor with the girl he adored
And the band played on
But his brain was so loaded it nearly exploded
The poor girl was filled with alarm
He married the girl with the strawberry curls
And the band played on

You've heard about the Thread Manager, the Mac OS's implementation of cooperative threading. You've read the *develop* articles. You've downloaded *Inside Macintosh:Threads*. You've seen the sample code. Now you want to build software to take advantage of this technology.

Threading really shines when your software spins off a lengthy task and returns control to the user immediately. Instead of having to wait for your software to complete the command, your user is free to continue working.

One of the main bottlenecks that software faces isn't computational speed, it's input/output (IO) speed. Since IO tends to take so long, it's an ideal candidate for threading.

In this paper I'll cover:

- What IO means
- The Mac OS's IO programming interface with examples
- How to match the Thread Manager with Mac OS IO
- The window of death (PG-13 — may be unsuitable for developers under 13 years of age)
- How to effectively couple IO with the Thread Manager

IO Overview

IO is by definition the act of moving data from an IO device to RAM (input) or from RAM to an IO device (output). Input also goes by the less formal name "read" while output goes by "write."

Common examples of IO devices are:

- SCSI devices like hard drives, scanners and CD-ROM drives
- Serial devices like modems, printers and other Macs
- ADB devices like mice and keyboards

So your hard drive, modem and keyboard all work towards the same noble goal of blasting bits to and from RAM. Humbling, isn't it?

The Mac OS IO Programming Interface

All this hardware stuff is fine and dandy, you say, but I'm a software guy. How do I code this stuff? I'm glad you asked, otherwise this paper would be rather short.

Like most other operating systems, the Mac OS divvies up the task of managing IO devices. Sitting right above the hardware are chunks of code called **drivers**. Their job is to provide a software interface for the hardware. By abstracting the hardware through drivers, you don't have a bunch of software touching the hardware willy-nilly — all access goes through one channel. If the hardware changes, only the driver needs to be rewritten.

In order to manage these drivers and the devices they control, Apple devised the **Device Manager**. Your application uses the Device Manager to handle IO — you rarely talk to drivers directly.

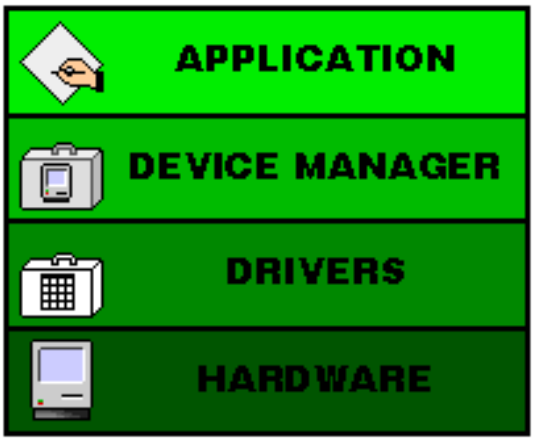

Figure 1: The Layering of the MacOS IO Programming Interface

To understand Mac OS IO is to understand the Device Manager. Fortunately, IO isn't a complex topic, and neither is the Device Manager. In fact, the entire Device Manager programming interface is just a few variations on seven basic commands: **Open**, **Close**, **Read**, **Write**, **Control**, **Status** and **KillIO**.

The Open command is used to open a connection to a driver. To be nice, make sure you call Close when you're done using the device.

The Read command is used to move data from the device into RAM. The Write command is used to move data from RAM to the device. These are the meat of the Mac OS IO programming interface.

Control is used when issuing command not directly related to pumping data. Changing a serial port's speed, for example.

Status is the flip side of Control, you can use it to get a serial port's speed.

KillIO has a special purpose that we'll get into momentarily.

To execute an IO action, you create an **IO job**. A job is simply a description you pass to the Device Manager of the IO task you'd like accomplished. Some information included in an IO job are the source of the data to transfer, the destination and the size of the transfer.

There are two models for executing IO jobs: synchronous and asynchronous. In a nutshell, the synchronous model is easy to code but locks up your Macintosh until the IO job completes. The asynchronous model is more difficult to code but doesn't lock up your Macintosh until the IO job completes.

Fortunately, when you combine an asynchronous model with the Thread Manager, you get a new model, **threaded IO**. Threaded IO combines the synchronous model's ease of use with the asynchronous model's parallelism. A worthy goal indeed.

The Synchronous Model

The synchronous model for executing IO jobs is easy to code. Each of the Device Manager commands (Open, Close, Read, Write, Control, Status and KillIO) are represented by one function call.

It could scarcely be easier to use the synchronous model — the IO job is specified in the parameters of each function. Let's look at the function prototypes:

```
OSErr    OpenDriver( ConstStr255Param name, short *drvrRefNum );

OSErr    CloseDriver( short refNum );
OSErr    FSRead( short refNum, long *count, void *buffPtr );

OSErr    FSWrite( short refNum, long *count, const void *buffPtr );
OSErr    Control( short refNum, short csCode, const void *csParamPtr );

OSErr    Status( short refNum, short csCode, void *csParamPtr );
OSErr    KillIO( short refNum );
```

Everything seems in order here. When you want to use a driver, you call `OpenDriver()` specifying the name the the driver in question. If all goes well, you get a **reference number**

passed back in `drvrRefNum`. A reference number is a unique ID you use when referring to an open driver. Notice every other function takes a variable named `refNum`.

Once you're done with the driver, call `CloseDriver()` with the aforementioned reference number that `OpenDriver()` gave you.

You use `FSRead()` to read. You pass it the omnipresent reference number, the size of the job (`count`) and where in RAM to put the read data (`buffPtr`).

`FSWrite()` is just like `FSRead()` except `buffPtr` now points where to get the data to write out instead of where to put the data.

When you want to pass a Control message to a driver, you call `Control()` with constant in `csCode` that maps to the message you're passing. For example, the change serial speed message constant is 13 (`serdSetBaud`), so we'd set csCode to 13 to change the serial port's speed.

The `csParamPtr` argument for `Control()` is where you stick the information relevant to the Control message. In the serial speed scenario, we'd set `csParamPtr` to point to a `short` that tells that Serial Driver what speed to set the port.

`Status()` is `Control()`'s mirror twin. Use it to get information from the driver. `csCode` and `csParamPtr` work the same way as with `Control()` except the information is now outgoing instead of incoming.

KillIO() deals with asynchronous IO — we'll talk about it then.

A Synchronous IO Example

To illustrate the various models (synchronous, asynchronous, threaded), we'll code the same simple task to each model. The simple task is to write the 4 byte string `ATZ\r` to the modem port. For those of you who don't know, `ATZ\r` is the modem reset command in the Hayes' AT command set. Assuming a modem is attached to the modem port, the modem will reset itself.

Before we get into the code, let me note a Serial Driver quirk. Each serial port is controlled not by one but by two separate drivers: an input driver and an output driver.This separation is a work-around for a Device Manager constraint.

There's a few things to remember. One, the output driver is the dominant driver. Open it first, close it last, send all Write, Control and Status commands to it. Two, the Read command should only be directed to the input driver. Three, only the still-mysterious KillIO command can be directed to both the input and output drivers.

Here's a function that uses the synchronous model to execute our sample IO job:

```
OSErr    SynchronousModemReset()
{
    Str255   resetCmd = "\pATZ\r";
    short    inRefNum = 0, outRefNum = 0;
    long     count = resetCmd[ 0 ];
    OSErr    err;

    /*    Attempt to open the modem serial port */
    err = OpenDriver( "\p.AOut", &outRefNum; );
    if( !err )
        err = OpenDriver( "\p.AIn", &inRefNum; );

    /*    Write the modem reset command using the synchronous model */
    if( !err )
        err = FSWrite( outRefNum, &count, resetCmd + 1 );

    /*    Call the test function */
    if( !err )
        Foo();

    /*    If we successfully opened the modem serial port, close it now */
    if( inRefNum ) {
        (void) CloseDriver( inRefNum );
        inRefNum = 0;
    }
    if( outRefNum ) {
        (void) CloseDriver( outRefNum );
        outRefNum = 0;
    }

    return( err );
}
```

First we initialize five variables: `resetCmd`, `inRefNum`, `outRefNum`, `count` and `err`. `resetCmd` holds a Pascal string containing the `ATZ\r` command. `inRefNum` and `outRefNum` will hold the input driver's reference number and output driver's reference number, respectively. Until then, we initially set them to zero. We do this to mark the reference number as invalid. Bad things happen if we attempt to use an invalid reference number. `count` holds the size of the IO job. Finally, `err` holds the error code.

First we open the output driver and snatch its reference number. If that works, then we open the input driver.

If we are able to open both drivers then we write the modem reset command string out the modem port. `Foo()` will then be called once `FSWrite()` successfully returns.

We're all done here, now we make sure the input reference number is valid before charging off to close the driver. We ignore the error code returned by `CloseDriver()`, because there's nothing we could do about it if it failed.

Note we invalidate `inRefNum` by setting it to zero after we're done with it. This is a good precautionary measure to take. We then close the driver with the `outRefNum` reference number.

Synchronous IO Drawbacks

Synchronous IO has two drawbacks. Your computer is effectively frozen while the IO job completes. Interrupts are still handled, however anything depending on `WaitNextEvent()` is cut off. It's as if one process is hogging the processor. This is a bad thing.

The second drawback stems from the first: there's no way of handling timeouts. Our modem reset command is a good example. What if the modem isn't connected to the modem port when we synchronously write `ATZ\r` to it? We wait forever for the IO job to complete — hanging the computer. Unfortunately the only way to discover if something is plugged in is to blindly write to the port.

The way out of this is to write our modem reset command and wait maybe 7 seconds. If we didn't execute our IO job by then, it's an indicator that nothing is attached to the serial port.

The Asynchronous Model

The asynchronous model is a low-level programming interface — you have to do extra work to use it, but it's more flexible.

Whereas you specify your IO job in the synchronous model's function parameters, you specify your IO job in **parameter blocks** when using the asynchronous model. A parameter block is simply a struct. When calling a function that uses a parameter block, you pass along a parameter block's address as the argument.

Specifying IO jobs in a parameters block is complex and error-prone, but you gain three advantages. First, when dealing with this many parameters, it's difficult to fit them all into a function's argument list. Second, it's easy to extend the structure to add your own fields. Third, and most important, by giving the parameter block its own chunk of memory, you can make it queueable.

Let me clear up that last statement. The Mac OS has a set of utilities named the Queue utilities. The Queue utilities are functions that maintain linked lists. A linked list with a little extra information tied to it is called a **queue**.

Each driver has a **job queue** associated with it. A job queue is a linked list of parameter blocks. When you execute an IO job asynchronously, the Device Manager places the parameter block at the end of that driver's job queue instead of executing it immediately (like the synchronous model does). The Device Manager immediately returns control to your software.

Using interrupts, the driver completes the IO jobs in its job queue. Seemingly in parallel your IO job completes and is retired.

Now is a good time to fill you in on KillIO. Since you are given back control immediately after executing an asynchronous IO job, you may find yourself wanting to stop an IO job thats pending or currently executing. That's what KillIO does — it removes each pending IO job from the job queue and halts the current job. It's great for stopping run away IO jobs like our modem reset command string.

Here's the asynchronous programming interface's function prototypes:

```
OSErr    PBOpenAsync( ParmBlkPtr paramBlock );
OSErr    PBCloseAsync( ParmBlkPtr paramBlock );

OSErr    PBReadAsync( ParmBlkPtr paramBlock );
OSErr    PBWriteAsync( ParmBlkPtr paramBlock );

OSErr    PBControlAsync( ParmBlkPtr paramBlock );
OSErr    PBStatusAsync( ParmBlkPtr paramBlock );

OSErr    PBKillIOAsync( ParmBlkPtr paramBlock );
```

The basic commands are all here: Open, Close, Read, Write, Control, Status and KillIO. Drivers can't be opened, closed or killed asynchronously, so that just leaves us with `PBReadAsync()`, `PBWriteAsync()`, `PBControlAsync()` and `PBStatusAsync()`.

All the functions take the same argument type: `ParmBlkPtr`. *Inside Macintosh:Devices* tells us that `ParmBlkPtr` is a pointer to a `ParamBlockRec` union:

```
union ParamBlockRec {
    IOParam          ioParam;
    FileParam        fileParam;
    VolumeParam      volumeParam;
    CntrlParam       cntrlParam;
    SlotDevParam     slotDevParam;
    MultiDevParam    multiDevParam;
};
```

The various fields are used for different purposes depending on what drivers you're working with. `ioParam` is for transport drivers like the Serial Driver. `fileParam` is used for the File Manager. `volumeParam` is used for managing storage volumes like floppies, hard drives, CDs, etc. `cntrlParam` is used for controlling drivers themselves. We're most interested in the `ioParam` field and thus the `IOParam` structure:

```
struct IOParam {
    QElemPtr           qLink;
    short              qType;
    short              ioTrap;
    Ptr                ioCmdAddr;
    IOCompletionUPP    ioCompletion;
    OSErr              ioResult;
    StringPtr          ioNamePtr;
    short              ioVRefNum;
    short              ioRefNum;
    SInt8              ioVersNum;
    SInt8              ioPermssn;
    Ptr                ioMisc;
    Ptr                ioBuffer;
    long               ioReqCount;
    long               ioActCount;
    short              ioPosMode;
    long               ioPosOffset;
};
```

The first two fields, `qLink` and `qType`, are used by the Queue Manager. The next two fields, `ioTrap` and `ioCmdAddr`, are used internally by the Device Manager. I wouldn't mess with them.

`ioCompletion` is an important field. When the IO job is completed, the Device Manager calls the function pointer in `ioCompletion` if the field is not nil. The user-supplied function to be called when the job is completed is called a **completion routine**. Completion routines may be executed at interrupt time and are subject to interrupt time code restrictions such as they can't use the Memory Manager, handles, QuickDraw, etc.

When the parameter block is successfully placed into the job queue, the `ioResult` field is set to 1. When the IO job is completed, `ioResult` holds either 0 (noErr) or a negative error code. You can use this knowledge to test if an IO job is completed. If `ioResult` is less than 1, the IO job is finished.

We can safely ignore `ioNamePtr`, `ioVRefNum`, `ioVersNum`, `ioPermssn` and `ioMisc` for now. Read *Inside Macintosh:Devices* for these details.

`ioRefNum` holds the much ballyhooed driver reference number. `ioBuffer` points to the place to put the data if reading or the place to get the data if writing. You fill in `ioReqCount` with the transfer size you'd like — `ioActCount` tells you what you actually have. Finally, you set `ioPosMode` to the positioning mode (from the start, from the end, from the mark, etc) and `ioPosOffset` is where to find the data when reading or where to place the data when writing.

An Asynchronous IO Example

Now we'll code the modem reset command using the asynchronous model.

```
OSErr    AsynchronousModemReset()
{
    Str255          resetCmd = "\pATZ\r";
    short           inRefNum = 0, outRefNum = 0;
    ParamBlockRec   pb;
    OSErr           err;

    /*    Attempt to open the modem serial port */
    err = OpenDriver( "\p.AOut", &outRefNum; );
    if( !err )
        err = OpenDriver( "\p.AIn", &inRefNum; );

    /*    Write the modem reset command using the asynchronous model */
    if( !err ) {
        pb.ioParam.ioCompletion = nil;
        pb.ioParam.ioRefNum = outRefNum;
        pb.ioParam.ioBuffer = (Ptr) resetCmd + 1;
        pb.ioParam.ioReqCount = resetCmd[ 0 ];
        pb.ioParam.ioPosMode = fsFromStart;
        pb.ioParam.ioPosOffset = 0;

        err = PBWriteAsync( &pb; );
    }

    /*    Call the test function */
    if( !err )
        Foo();

    /*    Wait until the asynchronous job completes */
    if( !err )
        while( pb.ioParam.ioResult > noErr ) {}

    /*    If we successfully opened the modem serial port, close it now */
    if( inRefNum ) {
        (void) CloseDriver( inRefNum );
        inRefNum = 0;
    }
    if( outRefNum ) {
        (void) CloseDriver( outRefNum );
        outRefNum = 0;
    }

    return( err );
}
```

The driver opening code and driver closing code is directly swiped from `SynchronousModemReset()`. We introduce the parameter block here, `pb`. We initialize a total of six fields in the parameter block before calling `PBWriteAsync()`.

Unlike with `SynchronousModemReset()`, `Foo()` will now possibly be called **before** the IO job is completed. If we wanted to make sure the IO job is finished before calling `Foo()`, we could move it after the `while` loop.

Speaking of which, the while loop takes advantage of the state of `ioResult` to determine if the IO job is done yet. It does nothing while waiting, but you could easily slip some code in that does some work.

Enter the Thread Manager

While you're waiting for IO to complete, you'd like to get some other work done. Apple answered our desires to have a general task sharing mechanism by creating the Thread Manager.

Asynchronous IO and the Thread Manager sound like they go together like peanut butter and chocolate. Imagine you spawn a download thread. While your download thread waits for the slow modem, it gives time to other threads.

The Ideal Threaded IO Model

Ideally, you'd only need to add two lines of code to enable your asynchronous IO code take advantage of the Thread Manager.

```
OSErr    IdealThreadedModemReset()
{
    Str255          resetCmd = "\pATZ\r";
    short           inRefNum = 0, outRefNum = 0;
    ParamBlockRec   pb;
    OSErr           err;

    /*    Attempt to open the modem serial port */
    err = OpenDriver( "\p.AOut", &outRefNum; );
    if( !err )
        err = OpenDriver( "\p.AIn", &inRefNum; );

    /*    Write the modem reset command using the asynchronous model */
    if( !err ) {
        pb.ioParam.ioCompletion = NewIOCompletionProc( WakeUpCompletionRoutine );
        pb.ioParam.ioRefNum = outRefNum;
        pb.ioParam.ioBuffer = (Ptr) resetCmd + 1;
        pb.ioParam.ioReqCount = resetCmd[ 0 ];
        pb.ioParam.ioPosMode = fsFromStart;
        pb.ioParam.ioPosOffset = 0;

        err = PBWriteAsync( &pb; );
    }

    /*    Sleep until WakeUpCompletionRoutine fires and wakes us up */
    if( !err )
        SetThreadState( kCurrentThreadID, kStoppedThreadState, kNoThreadID );

    /*    Call the test function */
    if( !err )
        Foo();

    /*    If we successfully opened the modem serial port, close it now */
    if( inRefNum ) {
        (void) CloseDriver( inRefNum );
        inRefNum = 0;
```

```
    }
    if( outRefNum ) {
        (void) CloseDriver( outRefNum );
        outRefNum = 0;
    }

    return( err );
}
```

Wouldn't be great if after you execute the asynchronous `PBWriteAsync()`, you could stop the thread and depend on the completion routine to reawaken the thread?

It *would* be nice — but you can't.

The Window of Death

Between when you call PBWrite() and you call SetThreadState(), the IO job can and will complete, executing our IO job.

Ideally, the execution path taken is like this:

- Asynchronously Write (`PBWrite()`)
- Stop the thread (`SetThreadState()`)
- Completion routine fires, readies the thread

However, this path of execution is possible

- Asynchronously Write (`PBWrite()`)
- Completion routine fires, attempts to ready the currently ready thread
- Stop the thread (`SetThreadState()`)

Your thread is stopped and will never be readied. Your thread is dead!

develop's Coping Mechanism

develop, Apple's Technical Journal, had an article on the Thread Manager. They advocated a dual thread solution.

There's two threads per IO job: the IO thread and the waker thread. Here's its execution path:

- The IO thread stops the waker thread
- The IO thread executes the IO job and stops itself
- The completion routine readies the waker thread, which is in a known stopped state
- The waker thread readies the IO thread

This is a poor work-around. You have to manage two threads per IO job and the scheduling overhead is too great.

PowerPlant's Coping Mechanism

PowerPlant, Metrowerk's C++ framework, defers the completion routine.

PowerPlant uses the ideal threaded IO model with a twist. Instead of the completion routine blindly attempting to ready the thread, it checks to see if the thread is really stopped. If it's not, then PowerPlant sets a Time Manager task to execute 100 microseconds in the future. Hopefully by then the thread will be stopped.

This is a good work-around, however it complicates the completion routine.

The Polling Coping Mechanism

With the polling coping mechanism, the thread is never stopped. After executing the IO job, the thread simply polls `ioResult` until it's less than one, yielding all the while.

Surprisingly, due to the scheduling overhead, this method is as fast as PowerPlant's and doesn't require a completion routine. This is the best work-around. However, it is still a work-around and polling is inelegant — we want a solution.

Problems with the Coping Mechanism

By now you realize that the Thread Manager wasn't designed with IO in mind. We should be able to use the ideal thread model.

The latency of the work-arounds is too high. Imagine your application has 25 threads running. The IO thread executes an IO job and yields. Even if the IO job completes immediately, the IO thread will have to wait behind the 24 other threads before it runs again. And one of those threads is your event loop, which may switch out your application.

Extending the Thread Manager for Effective Threaded IO

Metaphysical question: what does it mean to stop a thread?

The Thread Manager thinks it means to mark a thread as ineligible for scheduling and schedule another thread.

My solution: Write a function that marks a thread as ineligible for scheduling **but doesn't reschedule**. This would put a thread into a known state before executing the IO job.

However, latency would still be high. When the IO job is completed we'd like our thread to be first in line. We'll also add the ability to mark a thread as "priority."

That's great! How do we do it?

Creating a Thread Queue

The Thread Manager provides a hook where you can install your own scheduler. However, the Thread Manager's data structures are completely opaque — there's no "thread queue" to access from our scheduler. You can't even access a reference constant given a `ThreadID`!

Even if we did install a custom scheduler, we wouldn't know what to schedule!

However there is a way — create and maintain your own thread queue. The Thread Manager provides three hooks meant for debugging: `DebuggerNotifyNewThread()`, `DebuggerNotifyDisposeThread()` and `DebuggerNotifyScheduler()`. We'll plug into these hooks to maintain three thread queues: an ineligible queue, an eligible queue and a priority queue.

Maintaining the Thread Queues

When our `DebuggerNotifyNewThread()` hook is called, we'll add an element to the eligible queue with the new thread's ID.

When our `DebuggerNotifyDisposeThread()` hook is called, we'll search our queues to find the element with a matching thread ID and remove it.

Finally, when our `DebuggerNotifyScheduler()` hook is called, we'll look at our priority queue. If there's a priority thread waiting we'll move it to the eligible queue and schedule it. Priority status should be fleeting — otherwise it will hog the processor. If there isn't a priority thread waiting, we'll just schedule the next thread in the eligible queue.

The Thread Queue Code

I've defined the XThreadElem structure to hold individual thread elements:

```
struct    XThreadElem {
    XThreadElemPtr     next;
    XThreadQueuePtr    queue;
    ThreadID           threadID;
};
```

`next` points to the next element in the queue. `queue` points to this element's owner while `threadID` holds (surprise!) the element's thread ID.

We'll store all three queues (ineligible, eligible and priority) in one handle as an array of XThreadElem structures. We'll use the standard Mac OS Queue Utilities to manage them. We'll keep the handle locked because the Thread Queue routine will be called at interrupt time.

Now we need a **queue header**. A queue header stores important information like the first element in the queue and the last element:

```
struct XThreadQueue {
    short             type;
    XThreadElemPtr    head;
    XThreadElemPtr    tail;
    XThreadElemPtr    mark;
};
```

The `type` field is there for Queue Utilities compatibility — we don't use it. The `mark` field points to the next thread to schedule.

I can't reprint all the Thread Queue code here — look at the included code if you're interested.

The Extended Thread Manager Programming Interface

In all, I define three extended Thread Manager calls:

```
OSErr           InitXThreads();
XThreadState    GetXThreadState( ThreadID threadID );
OSErr           SetXThreadState( ThreadID threadID, XThreadState state );
```

Call `InitXThreads()` once before calling any of the other extended Thread Manager calls. It allocates `XThreadElem` array and installs the Thread Manager debugging callbacks.

GetXThreadState() works like the Thread Manager's GetThreadState() except returns one of three constants:

```
enum {
    kXThreadIneligible = 0,
    kXThreadEligible,
    kXThreadPriority
};
```

SetXThreadState() works like the Thread Manager's SetThreadState() except it takes the extended Thread Manager constants and doesn't reschedule.

The Threaded IO Programming Interface

Now is when the rubber meets the road. We've extended the Thread Manager cleanly. Now we want to merge synchronous IO with the extended Thread Manager to give us easy-to-code high-performance IO.

Witness two new functions:

```
OSErr  ThreadedRead( short refNum, void *buffer, long *size, long offset, long
patience );
OSErr  ThreadedWrite( short refNum, void *buffer, long *size, long offset, long
patience );
```

`ThreadedRead()` and `ThreadedWrite()` are descendants of `FSRead()` and `FSWrite()`. They're more powerful, so follow along.

`refNum` is the standard reference number, `buffer` points to where to get the data or put the data. Set `size` to the size of the IO job — after the IO job is done `size` will be set to the actual number of bytes transferred. You specify where you want to read from or write to in `offset`. Finally, specify how long you're willing to wait in milliseconds in `patience`. One thousand milliseconds is equal to one second.

Enjoy!

All the code is included with this paper, hunt around and enjoy. I'm storing this paper at my web site and will continue to update it and the code. You can find it at: <http://www.u-s-x.com/wolfie/rants/andthebandplayedon.html>.

Bibliography

Apple Computer. *Inside Macintosh:Devices*. Addison Wesley, Reading, Massachusetts. 1994.