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.

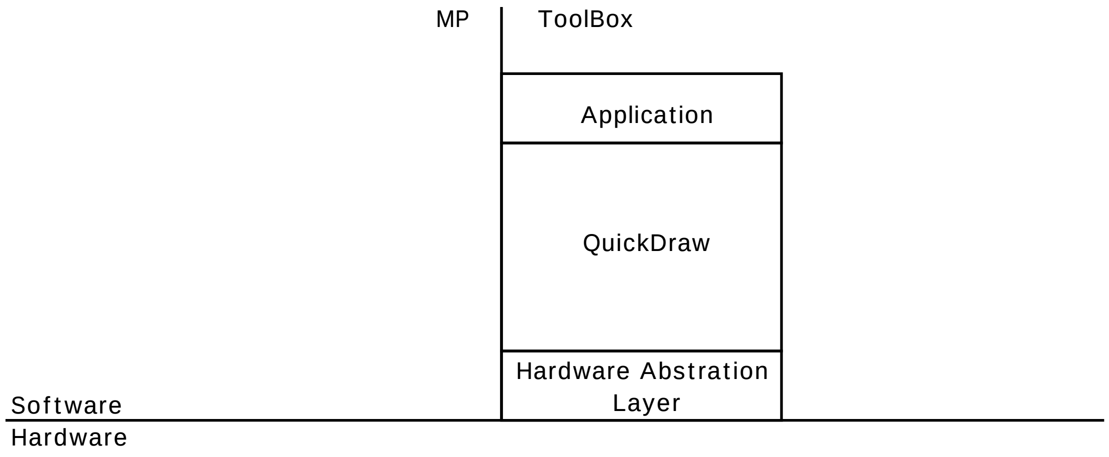


Figure 1: QuickDraw Direct Drawing

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.

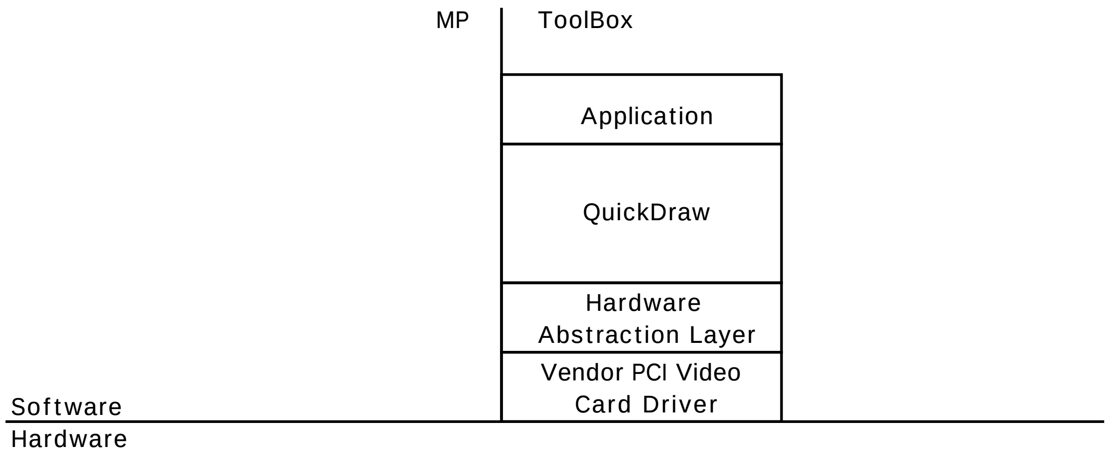


Figure 2: Accelerated QuickDraw Drawing

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.

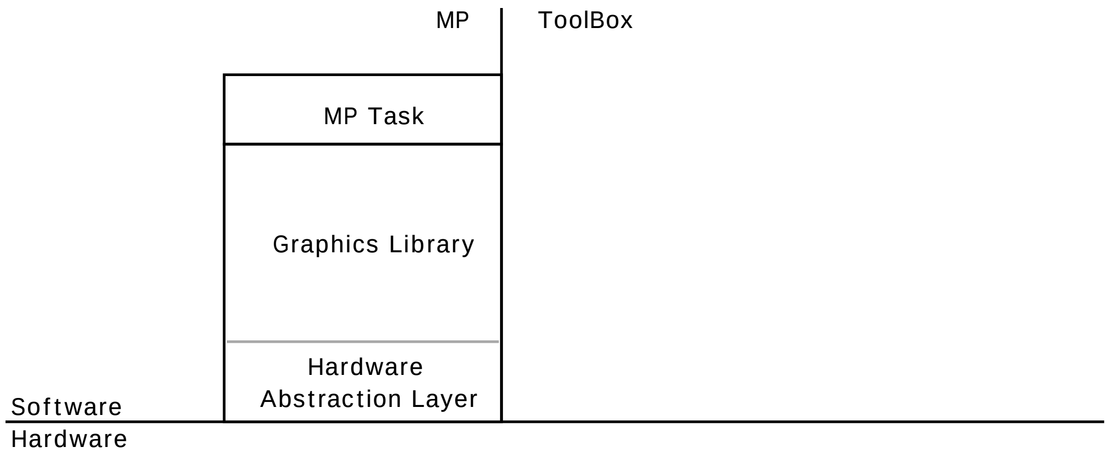


Figure 3: Direct to Screen MP Drawing

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.

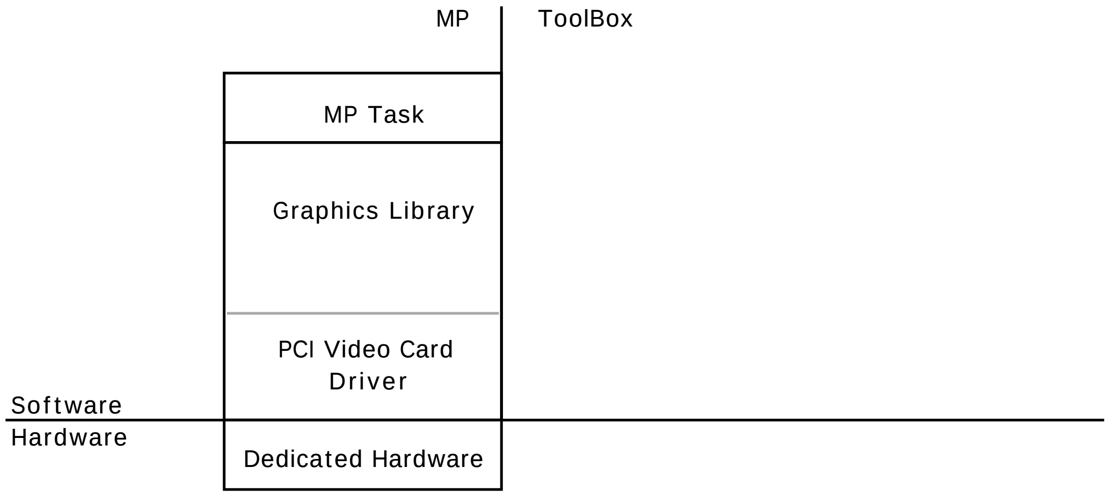


Figure 4: Accelerated MP Drawing

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.

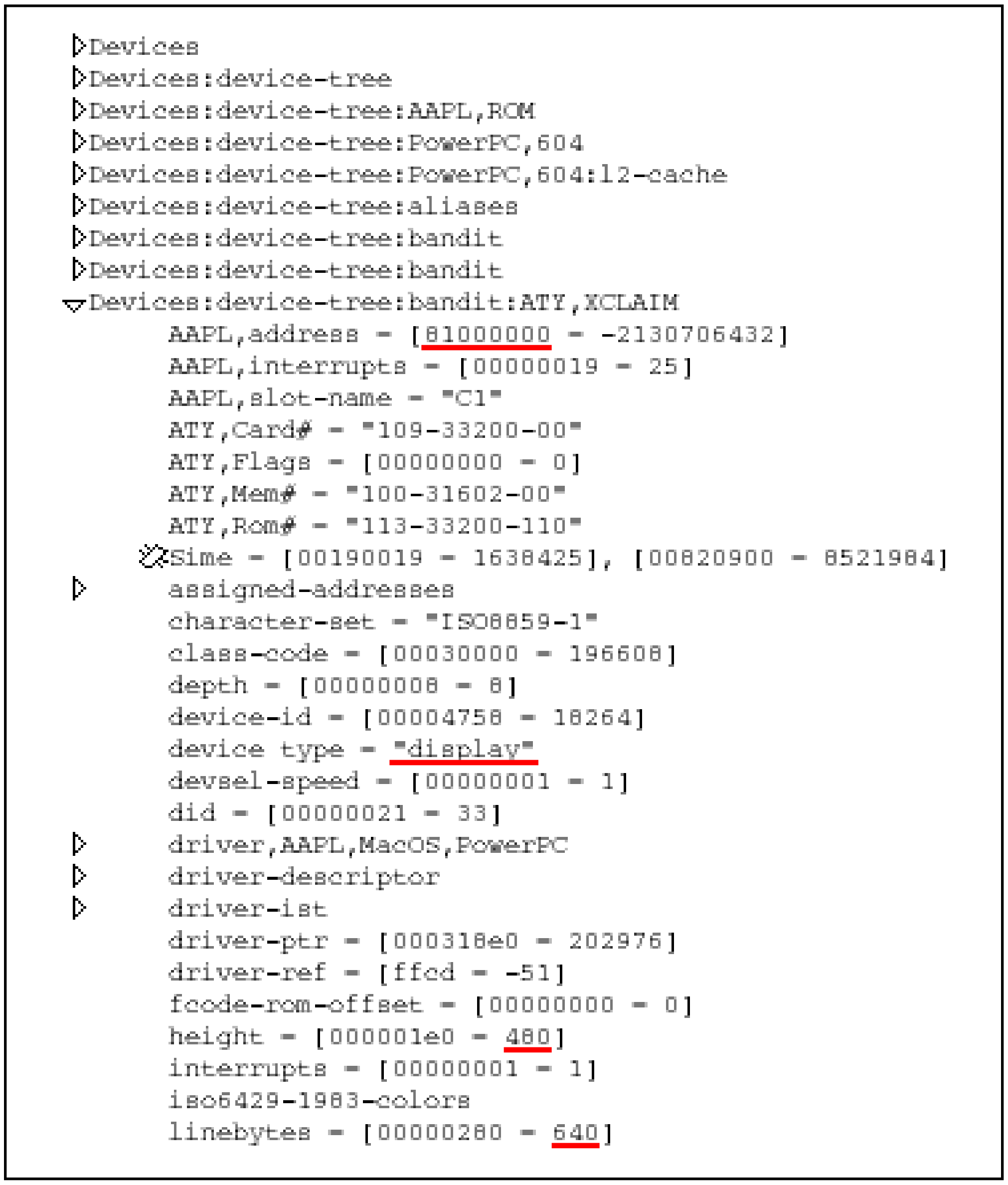
Step #1 - Finding the base address of the video buffer

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.



```
▷Devices
▷Devices:device-tree
▷Devices:device-tree:AAPL,ROM
▷Devices:device-tree:PowerPC,604
▷Devices:device-tree:PowerPC,604:l2-cache
▷Devices:device-tree:aliases
▷Devices:device-tree:bandit
▷Devices:device-tree:bandit
▽Devices:device-tree:bandit:ATY,XCLAIM
      AAPL,address = [81000000 = -2130706432]
      AAPL,interrupts = [00000019 = 25]
      AAPL,slot-name = "C1"
      ATY,Card# = "109-33200-00"
      ATY,Flags = [00000000 = 0]
      ATY,Mem# = "100-31602-00"
      ATY,Rom# = "113-33200-110"
      Sime = [00190019 = 1638425], [00820900 = 8521984]
▷     assigned-addresses
      character-set = "ISO8859-1"
      class-code = [00030000 = 196608]
      depth = [00000008 = 8]
      device-id = [00004758 = 18264]
      device type = "display"
      devsel-speed = [00000001 = 1]
      did = [00000021 = 33]
▷     driver,AAPL,MacOS,PowerPC
▷     driver-descriptor
▷     driver-ist
      driver-ptr = [000318e0 = 202976]
      driver-ref = [ffcd = -51]
      fcode-rom-offset = [00000000 = 0]
      height = [000001e0 = 480]
      interrupts = [00000001 = 1]
      iso6429-1983-colors
      linebytes = [00000280 = 640]
```

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_Rect     theClippingRect;

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.