So You Want Your Application to Fly!

NeXT Technical Support Notes

Written by Bruce Blumberg
Edited by Caroline Rose with Adrienne Wong

Special thanks to Leo Hourvitz, Trey Matteson, Paul Hegarty, and Matt Morse

Contents

Introduction

Nothing is slower than something that doesn't work.
— Henry Lieberman

I done seen everythin' when I seen an elephant fly.
— Dumbo

Many of you are at the stage where you're faced with the challenge of improving your application's real and perceived performance. The purpose of this document is to give you some suggestions on how to measure your application's performance, where to look to improve performance, and some techniques that we've found useful for improving real and perceived performance. In addition, since memory usage is closely tied to performance, we will touch lightly on how to measure memory usage and some techniques for using memory efficiently. The good news is that relatively small changes can often affect performance enormously, and much of this document is focused on those areas where there's a relatively big payoff. The bad news is that past a certain point, performance tuning is highly application-specific and generalities almost always fail. In a sense, this document is aimed at getting you to that point as quickly as possible.

This document assumes a working knowledge of the Application Kit™. Where topics are documented in *The NeXT™ System Reference Manual* or elsewhere, I will refer to the available documentation rather than repeat it here. (The *System Reference Manual* references apply to the preliminary 1.0 version of the documentation, either on-line or printed, unless noted otherwise.)

Lastly, this document represents an amalgam of personal experience, experiences of developers, and the experience and expertise of many of our engineers, in particular Leo Hourvitz, Trey Matteson, and Paul Hegarty. Any errors are of course contributed by yours truly.

Bruce Blumberg

Measuring Performance and Memory Usage

Where have all the cycles gone? Long time passing...
— Folk song

Measuring Performance

If you went to parochial school, you know that confession is the first step toward redemption. By just using your application you may already have a sense of where the problem lies—for example, load times are too long, redraw time is too long, or maybe response time is sluggish. In fact, this document is organized along those lines since specific techniques are often useful in solving each of those problems. However, there are a number of tools that can be used to get a qualitative and quantitative sense of where the cycles are being consumed within your application. The purpose of this section is to elaborate on them.

gprof

The standard profiling tool in the UNIX® world is **gprof**, and it's an excellent tool for understanding where in your process the cycles are being consumed. It generates a call graph for your application showing how many times a given function was called, and how much time was spent in that function and the function's descendents (that is, functions called by the given function). By analyzing the output from **gprof** you can quickly identify those areas where it makes sense to focus your optimization efforts. To use **gprof**:

- Compile the application using **make profile**. This compiles your program with the **-pg** option, which indicates profiling and generates an executable file named **appName.profile**.
- Run the application.
- When you exit, a file named **gmon.out** will be placed in your home directory.
- Run **gprof** to create the profile:

```
gprof appName.profile gmon.out >profile
```

- Use your favorite editor to examine the profile file.
- Or if you want to print the file, type

```
gprof appName.profile gmon.out |enscript
```

See the UNIX manual page on **gprof** for an explanation of the output of **gprof** and its many options.

However, there's one extremely important caveat with **gprof**: It does not measure the time spent by the Window Server on your behalf, because **gprof** only measures time spent within a given process. Effectively this means that **gprof** is not very useful in measuring time spent drawing, which is a serious problem because drawing time often accounts for the bulk of the total time. You must use other techniques to analyze the drawing time.

Qualitatively Measuring Drawing Performance

There are two command-line arguments that can be used to quickly get a qualitative sense of your drawing performance: NXShowAllWindows and NXShowPS.

NXShowAllWindows

The command-line argument NXShowAllWindows is extremely useful for identifying situations in which you're doing unnecessary drawing. NXShowAllWindows makes all your windows visible (including those holding Bitmap objects), and they're made retained so all drawing is done directly on the screen. Hence, by visual inspection, you can see all of the drawing that's going on in your application. NXShowAllWindows is often a good first step in diagnosing performance problems. To use NXShowAllWindows, launch your application from a Shell window, giving NXShowAllWindows as an argument:

```
appName –NXShowAllWindows YES
```

NXShowPS

The command-line argument NXShowPS is a useful aid in diagnosing inefficient drawing, because it writes all the PostScript® langauge code going to the Window Server out to **stderr** as well. While use of NXShowAllWindows will diagnose unnecessary drawing, use of NXShowPS will diagnose inefficient drawing. While the volume of PostScript langauge code is often a bit overwhelming, you should look for things like multiple erases, clips, and so on. To use this option, launch your program from a Shell window, giving NXShowPS as an argument:

```
appName -NXShowPS YES
```

For finer-grain inspection, use the **showps** and **shownops** commands within GDB to accomplish the equivalent of NXShowPS for selected portions of your application. In practice, this is more useful than using NXShowPS for your entire application, since you're typically interested only in the PostScript langauge code being emitted as a result of some method.

Quantitatively Measuring Drawing Performance

To get hard numbers on the time being consumed by an application in preparation for drawing, or by the Window Server on your behalf when it's executing drawing commands, you must use interval timing. There are two standard techniques:

- Bracket the relevant code with calls to **gettimeofday()** and calculate the elapsed time. Note that this will represent the "wall time," which is, after all, what the user cares about; it'll account for time spent within your application as well as within the Server.

- Bracket the relevant code with calls to **getrusage()** and **PSusertime()** to calculate the elapsed CPU time spent within your process, and the time spent within the Server on behalf of your application's Window Server context.

We've provided you with a simple Timing class (listed at the end of this document) which provides methods that implement the two techniques. To use the Timing class, you create an instance of the class, send a message to the object when you want to begin timing, and then send another message to it when you want to end timing. It'll keep track of the number of times the timing interval has been entered and the cumulative elapsed time.

Below are two methods from the Timing class that are used to calculate elapsed wall time.

```
-wallEnter
{
    cumTimesEntered++;
    NXPing();
    gettimeofday(&realtime,&tzone);
    synctime = realtime.tv_sec +realtime.tv_usec/1.0E6;
    return self;
}
```

```
-wallLeave
{
    double eTime;
    NXPing();
    gettimeofday(&realtime,&tzone);
    eTime = (-synctime + realtime.tv_sec
                + realtime.tv_usec /1.0E6) - tare;
    cumWallTime += eTime;
    return self;
}

-tare
{
    struct timezone tzone1;
    struct timeval realtime1;
    struct timeval realtime2;
    NXPing();
    gettimeofday(&realtime1,&tzone1);
    NXPing();
    gettimeofday(&realtime2,&tzone1);
    tare = (-realtime1.tv_sec + realtime2.tv_sec)+
        (-realtime1.tv_usec+ realtime2.tv_usec)/1.0E6;
    return self;
}
```

This approach is reasonably accurate, assuming the following:

- There are no other active processes on the machine (usually true).
- The unit of work being done is large enough that the round-trip overhead doesn't consume most of the time. The purpose of **tare** is to account for the round-trip time and measurement latency.
- You're not seriously affected by the timing variance this puts into your program (that is, don't do this in production code!)

The following two methods illustrate the approach of using **getrusage()** and **PSusertime()** to calculate the time spent within your process, and the time spent within the Server on behalf of your application's Window Server context.

```
-psEnter
{
    cumTimesEntered++;
    PSusertime(&stime);
    getrusage(RUSAGE_SELF,&rtime);
    synctime = (rtime.ru_utime.tv_sec+
        rtime.ru_stime.tv_sec) +
        (rtime.ru_utime.tv_usec +
        rtime.ru_stime.tv_usec)
        /1.0E6;
    return self;
}

-psLeave
{
    int et;
    double appTime;
    double psTime;

    getrusage(RUSAGE_SELF,&rtime);
    appTime  = ((rtime.ru_utime.tv_sec +
        rtime.ru_stime.tv_sec) +
        (rtime.ru_utime.tv_usec +
        rtime.ru_stime.tv_usec)
        /1.0E6) -synctime;
    cumAppTime += appTime;
    PSusertime(&et);
    psTime = ((et-stime)/1000.0);
    cumPSTime += psTime;
    return self;
}
```

In this case, **appTime** represents the elapsed CPU time spent in the process and in the system on behalf of the process, and **psTime** represents the CPU time spent in the Server on your behalf. The equation (**psTime**/(**appTime** +**psTime**)) will give you the percentage of the time spent in the Server.

The second approach is very useful for seeing the breakdown between application time and Server time within a chunk of code. For example, you can use this to answer questions like, “While wrapping and drawing a page of text, how much time is spent in the Server and how much time in the application?” You can then decide whether to go after the Server time with **showps** and timing your graphics operations, or go after the application time with **gprof**.

Either technique is extremely useful in understanding where the time is going, so you can then focus on solving the relevant problem. With either approach you should be sure to average over many trials to get an accurate value. You should also be sure to remove all timing code from production code.

Measuring Memory Usage

When measuring memory usage, you need to measure not only the amount of memory consumed by your process, but also the amount of memory consumed by the Window Server on your application's behalf. The tools described in this section allow you to measure both.

NXWindowServerMemory()

NXWindowServerMemory() provides an excellent mechanism for finding out how much Window Server VM and window backing store has been allocated by the Window Server on your application's behalf. While **NXWindowServerMemory()** does not measure performance per se, it's a very relevant tool to assist in performance tuning, since one of the frequent optimizations is to use off-screen windows or Bitmaps to cache frequently performed drawing. This technique increases performance at the expense of memory allocated in the Window Server. By using **NXWindowServerMemory()**, you can determine exactly what that cost is.

It pays to keep track of the amount of memory consumed by your application in the Server, particularly if you're using Bitmap objects and/or off-screen windows to cache images. Even a 3"-by-3" image with alpha consumes 40K, so memory usage can soar if you're making extensive use of Bitmaps. Be sure to free windows or Bitmap objects that are no longer needed or reuse them if possible. Also avoid using NX_UNIQUEBITMAP unless the Bitmap object needs to be in its own window.

At the end of this document is the listing for **NXWindowServerMemory()**, which describes the function and its use in more detail. It should be noted that **NXWindowServerMemory()** takes some time to be executed and so should not be used in production code.

ps

The UNIX utility **ps** is useful for determining how much memory is consumed by your process. However, interpreting the numbers can be a bit tricky. For example, every program has a large stack allocated to it, but only the pages that are actually touched ever come into existence in RAM or in the swapfile. They're created zero-filled on demand. However, the total amount of virtual memory allocated for your stack shows up in your virtual process size. Some or all of the shared libraries also appear in your virtual and resident sizes. Mapped files can also increase virtual sizes without increasing the resident size appreciably. The best use of the virtual size is finding substantial VM leaks. For most other purposes, resident size is what counts.

The **ps** utility is particularly useful in diagnosing large memory leaks. By running **ps -ux** in a Shell window at periodic intervals, and comparing the virtual and resident sizes of your application at the two points in time, you may be able to identify large memory leaks within your application.

size

The UNIX utility **size** is useful for determining the size of the various portions of your executable file (text, data, interface segments, and so on) on disk. For more information, see the UNIX manual page for **size**.

Improving Load Times of the Application and Panels

Why do the problem set today when it's not due until Friday?
— Former student

Often load times can be dramatically improved by taking a lazy approach to the creation and initialization of data structures and objects in your application. That is, don't spend launch time creating and initializing objects unless they're central to the working of your application, or unless they'll benefit the typical user and you want to optimize response time. Panels are a classic example of this.

In general, panels should be put in separate interface sections of the Mach-O executable file so that interface sections are loaded on demand rather than at startup time. Even if the panel is deferred (that is, the creation of the underlying window is deferred until the window is about to become visible), the time required to read in the interface section and do initialization can add up. For example, if you have a Preferences panel or an Info panel that's invoked pretty rarely, don't put the panel in the main interface section. Rather, put it in a separate interface section that's loaded the first time the user chooses the Preferences or Info menu item.

In many instances it may make sense to create a separate class that owns a panel and is the target for the panel's controls. In this case the interface section is typically loaded within the class method responsible for creating an instance of the class. The OpenPanel, SavePanel, and PageLayoutPanel classes all use this technique. Once again, create the object only when you absolutely need to. See the discussion on modules in the "Interface Builder" chapter of the *System Reference Manual.*

As you may know, Interface Builder associates a name with every object within an interface section. When **loadNibSection:owner:** is invoked, the names are read in along with the objects themselves. Subsequently, **NXGetNamedObject()** can be used to return the **id** of an object associated with a given name. However, if you don't need this functionality, you can save time and memory by not having the names read into memory. If you use **loadNibSection:owner:withNames:** rather than **loadNibSection:owner:** and pass NO as the argument for **withNames:**, the names will not be read in.

Another consideration for panels is whether to make them one-shot or not, which determines whether the underlying window should be freed when the panel is closed. If you have a panel that's unlikely to be brought up multiple times, you may want to consider making it one-shot. If the panel is brought up repeatedly, don't make it a one-shot panel. Note that even in the case of a one-shot Panel, the Panel object itself and its contents aren't freed, so they must be freed explicitly if you no longer need them.

The discussion above assumes that you want to trade off response time for startup time. Whether this is the “right answer” depends a great deal on the specific situation. For frequently used panels, you may want to optimize response time at the expense of startup time, and in these situations you may want to include the panel within the main interface section.

Improving Drawing Performance

Looks great. So when are you going to be done?
— The Pope to Michelangelo on seeing sketches for the Sistine Chapel

In most applications, a large chunk of time is consumed by drawing. Indeed, the optimization of those sections of your application that perform drawing should probably be your first priority, since often relatively small changes to your code can result in dramatic improvements in the real and perceived performance of your application. This discussion is broken down into three topics: General rules to follow in a client-server world, efficient focusing, and specific techniques for optimizing the performance of your drawing code.

General Rules in a Client-Server World

As you know, NextStep® is based on a client-server model, and this has certain implications when you're writing your application.

Write Your Program in Objective-C, not in the PostScript Langauge

Underlying the design of NextStep is the assumption that the center of gravity for a NextStep application should be on the application side rather than the Server side, that is, applications in the NextStep environment should be written in Objective-C®, and not in the PostScript langauge. This was based on performance considerations, as well as on the view that given the characteristics of the respective languages and the state of the respective development and debugging environments, it was much easier to write and debug applications in Objective-C than it was to do the same things in the 000000PostScript language.

The bulk of your algorithms and data structures should be written in Objective-C and should avoid maintaining state in the Window Server. This point about state is important for a number of reasons:

- If your application maintains state or other data structures in the Server, you must either provide PostScript langauge procedures to manipulate that state data, or you need to make calls to the Server whenever your algorithms on the application side need to access or modify that data. On the other hand, if you maintain two copies of the state data, one on the application side and one on the Server side, you'll need to ensure that they're always in synch. In either case, you'll require communication with the Window Server, and that implies interprocess communication. You're far better off to avoid the problem entirely by not maintaining state information in the Server, and doing all the work on the application side. If you have wraps that require state information, pass the state information as parameters to the wraps.

- Another side benefit of avoiding a lot of functions and state information in the Server comes when you print. As you know, printing occurs in a different Window Server context than that of your running application. As a result, if you have any PostScript langauge procedures or state variables that are in the Server, you must ensure that they're downloaded to the printing context as well. While this certainly can be done, it's often easier to avoid the problem altogether.

It should be noted that the above discussion shouldn't be construed as an argument against using wraps. In fact, replacing calls to multiple single-operator C functions with a single call to a wrap may result in a performance gain. Adobe recommends that while single-operator C functions are easy to use, wraps should be used for all but the simplest drawing. However, avoid using wraps for any operation that can be done entirely within the client application. It almost always pays to perform calculations on the application side and pass the results as parameters, as opposed to performing the calculations in the Server. Similarly, pass required state information as parameters to the wrap. Note also that there's little or no performance gain from downloading a wrap that contains a procedure definition and invoking the downloaded procedure from another wrap when needed, as opposed to downloading the procedure each time it's to be executed. Lastly, and perhaps most important, being smart about the drawing your application does and your choice of techniques to perform that drawing will probably have a much more dramatic effect on overall performance than the decision whether to use single-operator C functions rather than custom wraps.

Avoid Round Trips

Any PostScript langauge function or wrap that has a return value in its parameter list will cause a round trip to the Window Server, and should be avoided if possible. The most important thing to keep in mind is that every call to the Server requires interprocess communication and a number of context switches. This is always going to be slower than inspecting a variable in your application, or calling a method or function within your application. Thus, be judicious when making calls to the Server; consider whether by caching data or state on the application side you can avoid making the call altogether.

Use the Application Kit's Font Mechanism

The Application Kit provides a number of classes (Font, FontManager, and FontPanel) that can perform many font operations without interacting with the Server, thereby reducing the number of round trips. Wherever possible, you should use the font mechanism built into the Application Kit rather than interact directly with the Server. For example, the font mechanism caches font metric information on the application side. Hence, you can query the Font object for font metrics without involving communication with the Window Server. In fact, it's almost twice as fast to use the **getStringWidth:** method provided by the Font class than to use **PSstringwidth()** to get the width of strings before drawing.

Avoid Unnecessary Calls

Below is an example of making a call to the Server that's silly and inefficient. In this case you're better off just setting the gray, rather than querying the Server first for the current gray.

```
float currentGray;
PScurrentgray(&currentGray);
if(currentGray != desiredGray)
PSsetgray(desiredGray);
```

In general, you should avoid making unnecessary calls to the Window Server. Try to structure your drawing code so that you avoid redundant or unnecessary calls to **PSsetgray()**, **PSsetlinewidth()**, **PSgsave()**, and so on.

Boxcarring

The Display PostScript system has added a number of imaging operations that don't alter the underlying imaging model, but which cover frequently performed operations and provide highly optimized execution. These operations include:

user paths — A number of operators are provided that allow you to download a very compact representation of a path. The operator builds a special kind of path from it and performs some operation like stroke or fill on the path. (The use of user paths is discussed at length below.)

rectangles — A number of operators are provided to build a path made up of one or more rectangles, and perform some operation on the resulting path (fill, stroke, or clip). The operators can take either a single rectangle or an encoded array of rectangles. If you're doing any rectangle operations, you're almost always better off using these operators than **moveto**, **lineto**, and so on. Note the that Application Kit provides a number of functions (**NXRectClip()**, **NXRectClipList()**, **NXRectFill()**, **NXRectFillList()**, **NXEraseRect()**, **NXStrokeRect()**, and so on) that use the underlying rectangle operators provided by the Display PostScript system. You should use them whenever performing rectangle operations.

text drawing	A number of operators are provided that take an encoded array of coordinates and a character string, and use the coordinates to place and show the characters in the character string. (The use of **xyshow** is discussed below.)

We call the use of these operators "boxcarring" because they allow you to download a chunk of data to the Server, where it's operated on extremely efficiently. In fact, the use of these operators can be several times faster than the equivalent single-operator C functions or wraps. This is because the operands to these operators are always literals. In addition, the operators apply to the current graphics state and don't typically modify it (that is, the user path and rectangle operators don't affect the current path or point). In the case of user paths, the operator is passed some additional information that aids the interpreter in building the user path. As a result, the interpretation and execution of these operators by the Server can be optimized.

Needless to say, you should become familiar with these operators and use them wherever appropriate. They're documented in "PostScript Language Extensions for the Display PostScript System," available from Adobe Systems Incorporated (and included in NeXT's Release 0.9 technical documentation). Note also that the rectangle and user path operators above are simulated in the default print packages, so you can use them when printing without modification. However, **xyshow** and its relatives aren't simulated in the print packages, so you must use an alternative when printing. Because of their importance, user paths are discussed at length in a later section.

Efficient Focusing and gstate Objects

Before an application draws in a view, the view must be the **focusView** for the application, otherwise the drawing will appear in places other than where you want it. The process of making the view the **focusView** for the application is called "focusing." When you draw within the context of **drawSelf::**, focusing is performed by the view's **display** method. If you draw outside the context of **drawSelf::**, you manage focusing explicitly via **lockFocus** and **unlockFocus**.

Ensuring that your view is the **focusView** before drawing is a necessary step in drawing, but you should avoid unnecessary focusing. When the view is focused on, an appropriate transform matrix must be built to map the view's coordinate system back to the window's coordinate system, an appropriate clipping path must be built, and a **PSgsave()** is performed. While care has been taken to make this process as efficient as possible, it nonetheless takes time and is worth paying attention to. Here are some suggestions:

- The fastest focus is the one you don't need to do. Try to structure your code so that you aren't doing unnecessary focusing. For example, if you're drawing within the context of a modal loop, put the **lockFocus** and **unlockFocus** outside the loop.

- Turn off clipping if not needed. Building the clipping path is usually the slowest part of focusing, so you can usually save a good deal of time by turning clipping off for the view if you don't need it. You don't need clipping if you're sure you'll never attempt to draw outside the bounds of your view.

- Use **gstate** objects. (More on this below.)

If you have a view that's repeatedly being focused on (for example, the gauge on an instrument panel of a flight simulator), you should consider allocating a **gstate** object for the view via the **allocateGState:** method of View. A **gstate** object is a PostScript langauage object (not an Objective-C object) that contains all the information contained in a graphics state, and is an extension provided by the Display PostScript system designed to make it much more efficient to switch graphics states. The very first time a **lockFocus** or **display** is sent to a view for which a **gstate** object has been allocated, the view machinery focuses on the view in the normal manner, has the Window Server take a snapshot of the graphics state, and then sticks it in a **gstate** object. Thereafter, whenever focusing occurs, the view machinery first checks to ensure the **gstate** object is still valid (that is, the coordinate system of the view or its superviews has not been permanently changed) and, assuming it is, has the Window Server copy the **gstate** to the current graphics state.

Note: View has a handy method called **initGState** that's called automatically whenever the **gstate** is created or modified. You can override **initGState** to initialize other aspects of the graphics state, such as line width or gray level. Use the **allocateGState:** method of View to allocate a **gstate** for a view.

The use of **gstate** objects effectively avoids all the work normally associated with focusing, and subsequently reduces the time required for focusing. However, they don't come for free, and thus should be used judiciously. Since a **gstate** object contains all the information contained in a graphics state, including the current clipping path and current path, they consume a few hundred bytes at a minimum, and can be much larger depending on the nature of the **clippath** and **currentpath**.

If your application repeatedly focuses on and draws in a view, it may be a good candidate for a **gstate** object. On the other hand, if you have a view that your application focuses on infrequently, it may not make sense to allocate a **gstate** object for it, regardless of the amount of drawing you actually do in that view. The reasoning here is that allocating a **gstate** affects only the focusing time, as it has no effect on the drawing time once the graphics state has been set up.

You'll see less of a gain from using **gstate** objects if clipping has been turned off for the view. This is because building the clipping path is normally the most time-consuming part of focusing. For example, in the case of ScrollView, clipping is turned off for the ScrollView and its scrollers, and consequently no **gstate** object is allocated for them. On the other hand, a **gstate** object is allocated for the **contentView**.

There's another reason why no **gstate** is allocated for the scrollers in the previous example. In general, you don't need to allocate a **gstate** object for a control, because any control with a modal loop will typically focus once at the beginning of the loop and unfocus at the end. When the action message of the control is sent to the control's target, the target will typically do some drawing in response and bracket its drawing by a

lockFocus and **unlockFocus**. However, the focus is still on the control when the action message returns (courtesy of the target's **unlockFocus**). Thus, while it may make sense for the target of the control to have a **gstate** allocated for it, it's probably not worth allocating a **gstate** for the control itself.

Lastly, since every window has a **gstate** object allocated for it by default, you may not see much of a gain from allocating a **gstate** for the **contentView** of a window.

The key point here is that the use of **gstate** objects is a convenient and powerful way to reduce focusing time, but you should use them judiciously.

Efficient Drawing

In many cases, your application's overall performance will be determined by its ability to perform the minimal amount of drawing necessary and to use the most efficient means of performing that drawing. This section first describes how to avoid unnecessary drawing, and then discusses a number of the NeXT and extensions provided by the Display PostScript system that can radically improve drawing performance.

Avoid Unnecessary Drawing

The fastest drawing you'll ever do is the drawing you don't do. You should always look for ways to avoid doing drawing that's unnecessary. It's almost always worth the effort to do the work necessary to cull out drawing that's not visible, although you shouldn't take this to the extreme and implement your own two-dimensional clipping. Limit your drawing to the area within your view that's contained within the update rectangles passed to **drawSelf::**—they'll tell you the area within your view's coordinate space that's visible, based on the intersection of the view's frame and the frames of the views above it in the hierarchy. In the case of scrolling, they'll also tell you what area of your view needs to be redrawn. Even a simple approach of intersecting the bounding boxes of objects with the update rectangles, and drawing only those that intersect, will get you a long way. Of course, if you know that your view is wholly contained within its superviews and that your view is never going to be placed within a ScrollView, and you don't have graphical objects that live outside the frame of the view, then this is less important.

A related point is to avoid drawing in off-screen windows or panels until it's absolutely necessary. For example, if you have a panel that contains things that need to be updated as the user performs actions in another window, don't update the items unless the panel is visible or about to become visible. You'll note that menu updating is a common instance of this type of problem. In fact, the Application Kit provides a mechanism (discussed in a later section) to easily handle the updating of menus, panels, and windows. It should be noted that a consequence of this approach is that the time to bring a panel to the front will probably be longer than if the contents are updated as you go along. Whether this is an acceptable tradeoff depends on the specific situation.

Compositing

Compositing is probably the single most powerful mechanism for improving the apparent drawing performance, but it should be used wisely. Whenever you have a complicated image that must be redrawn repeatedly, you should consider creating a Bitmap object, rendering the image into the Bitmap object, and thereafter use compositing to put the image wherever you need it. If it's a complicated image, this approach can be many, many times faster than redrawing the image each time. This is the standard approach used for most types of animation and for dragging images.

Like most good things, compositing doesn't come for free, and the cost is the memory required for the Bitmap object (in fact, the Bitmap object itself lives in an off-screen window). While the Bitmap object tries to reduce the overhead, the fact remains that 2 bits/pixel without alpha or 4 bits/pixel with alpha is required. For example, a 3"-by-3" image with alpha represents almost 40K. In many instances, the performance benefit far outweighs the cost, but you should be aware of the cost. Also keep in mind:

- If you don't need transparency, don't use it. Having transparency doubles the storage requirements and has a negative impact on performance as well. An occasional mistake is to use the **clear** compositing operator to erase the background of the Bitmap rather than **NXEraseRect()**. If you "clear" the background, you're actually making it transparent with the attendant creation of an alpha channel.

- If you're compositing an image using NX_COPY, remember that you don't necessarily need to use **NXEraseRect()** before compositing, since the copy will obscure whatever is underneath anyway.

- Use NX_NOALPHABITMAP or NX_ALPHABITMAP as the type argument to the **newSize:type:** method of Bitmap, unless you explicitly want the resulting Bitmap in its own off-screen window. If one of these two constants is passed to the Bitmap object, it'll try to put the Bitmap in an existing window and thereby reduce overhead.

- Consider using user paths via **DPSDoUserPath()** as an alternative to compositing. User paths may be a good alternative if you have an image that's large but pretty simple, stroked, and intended to be put on top of another image. This last qualification implies that if you use compositing, you'll need to use transparency and the **sover** operator. In certain cases, user paths will be not only more efficient from a memory standpoint, but faster as well. Unfortunately there's no hard and fast rule here, but keep user paths in mind as an alternative.

- In situations where the user is dragging an opaque rectangular image over an arbitrarily complex background, it may pay to create a subclass of Window and override the **mouseDown:** method, so as the user drags the mouse within the window, the window will move. When you need to drag an image, you create an NX_PLAIN window of this class, composite in the image to be dragged, and, if necessary, remove the image from the background. Then, as the user drags the mouse, the image will move. When the mouse button comes up, you composite in the image into the background at the right position, and hide the drag window. This may be faster than just using compositing.

- Lastly, compositing isn't a supported operation during printing. If you use compositing to generate the on-screen image, you'll have to use an alternative (either the **image** operator or perhaps user paths) during printing, and consequently check NXDrawingStatus within the appropriate **drawSelf::** method to determine whether it's being invoked to image to the printer or to the screen. Note that if the use of compositing within the **drawSelf::** method is limited to compositing a pre-existing image from a Bitmap object using the NX_COPY operator, the Bitmap object will perform the appropriate conversion for you. Otherwise, you must explicitly provide an alternative means for rendering the image. In comparison, user paths are a supported operation during printing, and so you need not be concerned with whether the drawing is going to the screen or to the printer.

DPSDoUserPath()

NeXT provides a function called **DPSDoUserPath()** that facilitates the creation of a user path. The use of user paths via **DPSDoUserPath()** can be several times faster than using the equivalent single-operator C functions or wraps to draw even simple shapes. As a result, whenever you need to build a path of more than a few points (as few as four points or so), you may want to consider using **DPSDoUserPath()**.

DPSDoUserPath() relies on a new facility provided by the Display PostScript system called, not surprisingly, user paths (see the discussion of user paths in the Adobe documentation on Display PostScript system). A user path is a completely self-contained description (that is, containing only path construction operators and literal number operands) of a path in user space. Coupled with the fact that you're required to specify the bounding box of the user path as part of the user path, the interpreter can build a user path much more efficiently than a normal path. Since path creation by the Server is one of the more time-consuming tasks involved in drawing, this can be a very big gain. In addition, the Display PostScript system provides a mechanism to allow you to cache user paths to avoid redundant interpretation of the same path definition.

DPSDoUserPath() takes seven arguments:

- A pointer to an array of coordinates
- The number of coordinates in the array
- The numeric type of the coordinates
- A pointer to an array of path-building operators
- The number of operators
- A pointer to an array of coordinates that represents the bounding box for the user path
- An operator to be applied to the user path

DPSDoUserPath() takes these arguments and sends an encoded user path down to the

Server, where it's interpreted.

Here's an example of using **DPSDoUserPath()** to build a user path consisting of a series of vertical lines and then have the resulting user path stroked. It also asks the Server to cache the resulting user path so it can be reused.

```
-drawLines2
{
    float a;
    float *cArray;
    char *oArray;
    float bbox[4];

    int oi=0,ci=0;
    int size =(int) (bounds.size.width/(interval));

    /* fill cArray and oArray */
    cArray = (float *)malloc(4*numLines*sizeof(float));
    oArray = (char *)malloc(2+2*numLines*sizeof(char));
    oArray[oi++] = dps_ucache;
    oArray[oi++] = dps_setbbox;
    for(a=0.0;a<bounds.size.width;a+=interval){
        cArray[ci++] = a;
        cArray[ci++] = 0.0;
        oArray[oi++] = dps_moveto;
        cArray[ci++] = 0.0;
        cArray[ci++] = 150.0;
        oArray[oi++] = dps_rlineto;
    }
    /* fill bbox */
    bbox[0] = 0.0;
    bbox[1] = 0.0;
    bbox[2] = bounds.size.width;
    bbox[3] = 150.0;

    DPSDoUserPath(cArray,ci,dps_float,oArray,oi,bbox,
                dps_ustroke);
    free(cArray);
    free(oArray);
}
```

To give you a sense of the power of user paths, the following table shows a comparison of using single-operator C functions versus **DPSDoUserPath()** to draw a varying number of vertical, zero-width black lines. In each case, the time given represents the cumulative wall time for doing 100 iterations.

# Lines	Case 1	Case 2	Case 3	Case 4
5	3.57	2.95	2.41	2.19
10	4.46	3.85	2.64	2.37
20	6.99	6.39	3.13	2.70
40	11.34	9.99	4.09	3.36

Case 1 Draw lines using **PSlineto()** and **PSmoveto()**. **lockFocus** and **unlockFocus** inside loop.

Case 2 Same as Case 1 except **lockFocus** and **unlockFocus** taken out of loop.

Case 3 Draw lines using **DPSDoUserPath()**, arrays **malloc**'ed, initialized, and freed inside loop. **lockFocus** and **unlockFocus** outside loop.

Case 4 Same as Case 3 except that user path is cached via **ucache**.

User paths are beneficial even for a small number of points, and the benefit becomes even greater as the number of points goes up. Note also how taking focusing out of the loop has a noticeable impact on performance.

In some cases, **DPSDoUserPath()** may be a good alternative to compositing. For example, if you have what amounts to a stroked path that you want to lay on top of other drawing, using **DPSDoUserPath()** may be faster than rendering the path into a Bitmap object with a transparent background and using NX_SOVER to composite the image into the destination. This will be particularly true if the path is relatively simple and the number of pixels touched by the path is small in comparison to the number of pixels.

The knowledge and use of compositing and user paths is essential to fast drawing performance on our system. Be sure to read the relevant descriptions in the *System Reference Manual* as well as the Adobe documentation on the Display PostScript system, and start using these techniques if you aren't already using them.

PSxyshow()

If you need to do drawing of the form

```
x y moveto
(a) show
x1 y1 moveto
(b) show
```

(that is, a sequence of **moveto**s and single-character shows), you should consider using **PSxyshow()**, **PSxshow()**, or **PSyshow()**. While not as generally useful as user paths, it can provide the same type of improvement in performance. **PSxyshow()** is the single-operator function for **xyshow**, which is an PostScript langauge extension provided by the Display PostScript system (documented in the Adobe documentation on Display

PostScript system). These operators take a character string and an array of coordinates, go through the character string rendering a character, and then use the next one or two coordinates (depending on the function) to explicitly position the next character.

Those of you drawing rulers should look at **PSxyshow()** or one of its relatives.

Other Drawing Tips

There are a number of other things to keep in mind that will help with drawing performance.

Zero-Width Lines

Use zero-width lines whenever you need top line-drawing performance. Zero-width lines are special-cased by the interpreter and are the fastest way to draw lines. Note that when printing, the line width should be set to some other size since zero-width lines can "disappear" on high-resolution printers like the Linotronic®. Adobe recommends using a line width of 0.15 units, which will be treated as zero-width for the purposes of drawing on the screen, but will result in lines that are 0.15 units wide on a printer.

Dashed Lines

Don't use these. Dashed lines using **PSsetdash()** are extremely time-consuming. Consider using user paths or compositing as an alternative.

NXImageBitmap()

Best performance will be found when the size of the source and destination are the same, the source image and destination image are each two bits deep, and the source and destination are non-rotated. Even so, **NXImageBitmap()** is much slower than compositing, so if you can use compositing instead, do so. For example, you may want to image once into a Bitmap object, and then subsequently composite into the destination as required.

PSgsave() and PSgrestore()

Avoid unnecessary **gsave**s and **grestore**s. In particular:

- Since part of the built-in focusing mechanism via either **display** or **lockFocus** and **unlockFocus** is to do a **gsave** and **grestore**, your drawing code in either **drawSelf::** or within a **lockFocus/unlockFocus** is already bracketed by a **gsave** and **grestore**, so there's no need for you to explicitly do it.

- When stroking or filling a path, don't put a **gsave** or **grestore** around the stroke or fill if you aren't planning to use the path again. As Glenn Reid points out in *PostScript: Language Program Design*, the following construction is particularly inefficient:

```
0 0 moveto 100 100 lineto
gsave stroke grestore newpath
```

- Also, if you're clipping to a path, remember that the clip doesn't affect either the path or the current point. Avoid the following construction:

```
gsave clip grestore
```

Improving Response Time

Go ahead...make my...uh...uh...day
— Original actor chosen to play Dirty Harry

Users are very sensitive to response time. Even a slight delay in providing the users with feedback to their actions will give the impression of sluggishness regardless of how "fast" your application really is. Therefore, it's worth paying attention to response time and looking for ways to improve it. The discussion that follows provides a few techniques for improving real and perceived response time.

Optimal Panel and Menu Updating

It's often the case that you must update the state of items contained within menus and panels to reflect the user actions within the application. This may need to be done after every event. How you perform this update can have an important impact on the responsiveness of your application. The key is to avoid unnecessary drawing, particularly in menus and panels that aren't visible to the user. After all, why update a menu or panel that isn't visible? The Application Kit has a built-in mechanism for allowing you to handle updates efficiently. We encourage you to take advantage of it.

The key idea is that windows and menus have an **update** method that's responsible for handling updates. This method is invoked in the following situations:

- Automatically when a window is about to be brought on-screen.
- Whenever the **updateWindows** message is sent to NXApp, **updateWindows** will send the **update** message to every on-screen window. If you send a **setAutoUpdate:YES** message to NXApp, **updateWindows** will be sent after every event.
- Whenever a Command-key event is processed, the **update** message is sent to each panel and menu before the panel or window sends the **performKeyEquivalent:** message down its view hierarchy. Note that in this case the panel or menu may not be visible, but the update is necessary so you can disable or enable things that may respond to the key equivalent.

While the default **update** method of Window and Panel does nothing but return **self**, the **update** method of Menu has been implemented to allow you to easily update menu items in response to an **update** message. To take advantage of this, you must do two things:

- The **setUpdateAction:** message needs to be sent to each menu item that will need to be updated, passing the method you want invoked when the menu containing the menu item receives an **update** message.

- NXApp or the delegate of the menu must implement the method specified above. The method must take a single argument which is the **id** of the menu cell, and return YES if the cell needs to be redisplayed, or NO if not.

Here's a simple example of an **updateAction**:

```
-updateIt:(id) menucell
{
    if([menucell isEnabled]!=enabledFlag){
    [menucell setEnabled:enabledFlag];
    return YES;
    }
    else
    return NO;
}
```

When the menu's **update** method is invoked, it first disables **flushWindow**, and then goes through its item list. For each menu cell for which an **updateAction** has been specified, it disables **display**, invokes the update action, and if the update action returns YES, **display** is reenabled and the cell is redrawn. If at least one menu cell has been redrawn, the menu is flushed. It also does a number of other things to ensure optimal redraw.

Since a menu's **update** method is invoked only when the menu is visible, is about to become visible, or is about to process a key equivalent, you can be assured that the menu item is updated only when absolutely necessary, and in the most efficient manner possible. Once again, this is an example of taking a lazy approach.

This is documented in the *System Reference Manual* in the section "Managing a Window's Display," in the "Program Dynamics" chapter.

The documentation also discusses a means of disabling **display** while updating either multiple views or multiple items within a view. It turns out that this can have an unpredictable effect on drawing performance in the 1.0 Application Kit. For example, in the case of Matrix, it's more efficient not to **disableDisplay** or **autodisplay** when updating multiple items with the matrix. The reason is that when an individual cell needs to be redrawn, it only draws the contents of the cell, whereas when the matrix responds to a **displayIfNeeded**, the borders of the cells are redrawn as well.

Optimal Window Flushing

When using buffered windows, you may want to think about your strategy for flushing the window. All drawing in a buffered window is done off-screen in the backing store for the window. The Window Server maintains a “dirty rect” that represents a rectangle which encompasses all the bits that have been touched in backing store since the last time the window was flushed. Only the area of the window contained in the dirty rect is flushed when the window is asked to flush itself. It’s only when the dirty rect is flushed on-screen that the user sees the new drawing. Flushing is performed automatically once a view and its subviews have been drawn, if you use the **display/drawSelf::** mechanism provided by the View class. However, if you use **lockFocus** and **unlockFocus**, you must explicitly flush the window. In either case, here are some things to think about in determining your flush strategy.

It may be desirable to flush the window at intermediate points if drawing the view (or the views) is going to take some time. For example, if you’re doing a page layout system and are paginating a page, you may want to flush the window after every line, rather than wait until the entire page is done. While the overall time may be no faster, the users’ perception will be that it’s faster, because they can see that things are happening. This can be a very powerful technique for enhancing perceived performance.

You may also want to do intermediate flushes if you’re doing localized drawing in opposite corners of your window. If you’re doing localized drawing in widely different places in the window, the dirty rect will encompass many bits that haven’t changed and don’t need to be flushed. In this case, you might be better off doing intermediate flushes, rather than waiting and doing a single flush at the end of all your drawing.

However, if you’re updating several views within a window, and they aren’t in the same branch of the hierarchy *and* you don’t want the user to see intermediate results (one view updated, another not), then you should send the enclosing window the **disableFlushWindow** message before drawing. At the end of drawing, send it the **reenableFlushWindow** and **flushWindow** messages to actually flush the window. The advantage of this approach is that the user doesn’t see intermediate results. The disadvantage is that if the draw time is significant, users may be waiting for a period of time to see the results of their actions.

Coalescing Key Events

Generally speaking, the Text object should be used for handling and drawing text, but if you're responding to key-down events yourself and doing drawing in response, it's a good idea to coalesce key events as the Text object does automatically. By this, we mean that when you respond to key-down events, check whether there are any more such events in the queue, and if there are, pull them off before responding to the event so you can process them all at the same time. The Draw demo application does this with the arrow keys (that is, if Draw can't keep up with the rate at which you're moving the selection via the arrow keys, it'll combine as many of the moves as it can into one, bigger move). Note that the system does this for you in mouse-dragged events.

A related idea is to periodically check within your drawing code to see whether another event has come in. For example, if the user types a character in the middle of a line that causes the rest of the paragraph to be rewrapped, then as the program goes off to rewrap that text, it can periodically check to see whether the user has typed another character, at which point the program can abort rewrapping and start wrapping from the insertion point again.

Both of these techniques can contribute to the user's perception of a more responsive application, even if the performance isn't any better.

Wait Cursor

Use the NXWait cursor to indicate that the application is busy performing some action, if that action will take some time. This will provide feedback to users and partially satisfy their need for responsiveness, even though all the application is saying is "wait a bit."

PSobscurecursor()

Use **PSobscurecursor()** to hide the cursor rather than **PShidecursor()** and **PSshowcursor()**. The response latency will be better, since the Window Server will make the cursor appear the moment the user moves the mouse, without requiring the intervention of the application.

NXPing()

Communication between the Window Server and the application is buffered at both ends. While this is done for efficiency reasons, it can cause unwanted latency, particularly when PostScript langauge code is sent to the Server at a faster rate than the Server can consume it. There's a function called **NXPing()** that can be used to flush the buffers on both sides of the connection. Specifically, **NXPing()** waits until all PostScript lanaguage code sent to the Server has been executed. Since **NXPing()** involves a round trip to the Server, its use lowers overall throughput, but in certain cases its use can reduce latency and thus improve perceived performance. If you're doing a minimal operation in your loop, like rubber-banding out a rectangle in response to a mouse-dragged event, **NXPing()** is unnecessary, as the Server can keep up with the volume of PostScript langauge code being sent to it on each iteration. However, if you're generating a lot of PostScript language code on each iteration, the Server may still be working on the PostScript language code from the last iteration when the application starts sending it PostScript language code from the next iteration. Using **NXPing()** in this case can help keep the application and the Server in synch. Note that any operator that involves a round trip to the Server will have the same effect as **NXPing()**.

No discussion of **NXPing()** would be complete without noting that its use is hotly debated. It's absolutely clear that its use will reduce overall throughput, and probably should be avoided as a matter of course. If, however, you're having latency problems, you may want to experiment with **NXPing()**.

Reducing Memory Usage

Virtual-smirtual, you can't use what you don't have.
— Memory salesperson

The primary focus of this document has been to address performance issues. However, a few notes are in order concerning memory usage, particularly as it relates to performance. Faced with what seems to be nearly inexhaustible amounts of VM, applications sometimes seem to grow to fill the available space, and then some. The result is a higher level of paging than might otherwise be needed. In some cases, memory usage is traded for better performance (use of off-screen windows and compositing come to mind). In other cases, memory is simply not used efficiently. Reflect on the number of instructions that can be executed in the time it takes to handle a page fault and to bring in a page from the disk, and you'll begin to appreciate the need to be careful about the use of memory.

The following are some tips concerning memory usage.

Bitmaps and Off-Screen Windows

The most common performance optimization is to create off-screen Windows or Bitmap objects, draw into them once, and then composite on-screen. As mentioned earlier, you should use this technique judiciously, since it can consume a lot of memory in the Window Server. Consider using user paths as an alternative when possible. If using Bitmap objects, avoid using NX_UNIQUEBITMAP unless necessary, and don't use alpha unless you need to. Perhaps most important, remember to free the Bitmaps or Windows when you're done with them, or reuse them if possible. Use **NXWindowServerMemory()** to get a handle on the amount of memory you're consuming in the Window Server. See the discussion of **NXWindowServerMemory()** earlier in this document.

Garbage Collection

Remember you're the garbage collector for your application. Objects will not get freed unless you explicitly free them. If you're done with an object, free it! Remember also that if an object's instance variables point off to other objects or **malloc**'ed memory, you may need to explicitly free them as well (that is, freeing an object doesn't necessarily free its instance variables). In general, the Application Kit classes free their instance variables as appropriate (for example, freeing a window frees all of the views within that window's view hierarchy). However, when you create a subclass of an Application Kit class and add instance variables, you'll need to override the **free** method for that class and do the following:

```
-free
{
/* free new instance variables */
/* . . . */
/* now do what your super class would normally do */
return [super free];
}
```

If your application seems to grow over time, check to be sure that objects are freed when no longer needed. Alternatively, see if you can reuse existing objects.

Use of Good Sizes and malloc()

If you allocate a lot of blocks of a certain size, you can help **malloc()** be more efficient in its allocation of memory with a function called **malloc_good_size()**. When **malloc()** is asked to allocate memory, it consults a list of "good sizes" and finds the smallest "good size" that's greater than or equal to the requested allocation. By default, **malloc**'s "good sizes" are 16, 32, 64, 128, 176, 252, 340, 508, 680, 1020, 1360, 2044, 2724, 4088, or any multiple of 8192 bytes. For example, if you ask **malloc()** to allocate a block of 1400 bytes, it'll actually allocate 2044 since that's the smallest predefined good size that's greater than 1400. If you tell **malloc()** to make 1400 a good size, you don't waste as much space. Use this only if you allocate enough of these chunks to fill up a couple of pages (8K).

The function **mstats()** can be invoked within GDB to print a summary of memory usage by **malloc()**, including the total amount of memory allocated and an efficiency value calculated from the ratio of the space available in the blocks to the total amount of memory allocated by **malloc()**.

See the UNIX manual page on **malloc()** for more information on **malloc()**, **mstats()**, and **malloc_good_size()**.

NXUniqueString()

The Application Kit provides a number of functions (**NXUniqueString()**, **NXUniqueStringNoCopy()**, and some others) that create unique strings which are allocated once and then can be shared. The unique strings created are of type NXAtom, which means that they can be compared via **==** rather than **strcmp()**. Creating the strings is also very fast because the strings are maintained in a hash table, so search time is kept to a minimum. Once created, the strings are essentially read-only and should not be deallocated.

Use of unique strings can pay off handsomely in cases where you have read-only strings and there's repetitive use of the same string. A perfect example of their use is in the mail summary window in Mail. The strings contained in the date and name fields should be instances of unique strings, since they're read-only and the same string is used many times.

For more information on unique strings, see the description of **NXUniqueString()** in the "C Functions" chapter of the *System Reference Manual.*

loadNibSection:owner:withNames:

As mentioned earlier, if you don't use the names generated by Interface Builder, you can reduce the amount of memory associated with reading in an interface section by using the **loadNibSection:owner:withNames:** method, passing NO as the argument for **withNames:**. See the earlier section "Improving Load Times of the Application and Panels" for more information.

Lazy Creation of Objects

You can reduce the working size of your application by not instantiating objects until you need them. This is particularly true if the object is used infrequently or not at all unless the user takes a particular action. By not instantiating the object until you need it, your application does not need to grow until it's absolutely necessary to do so.

On a related point, all windows that aren't visible at launch time should be set to be deferred. This means that when the interface section containing the window is loaded in, space will be allocated for the Window object, but no window will be created until the window is sent the **orderFront:** message. Thus, it's only when the window is about to come on-screen for the first time that the underlying window is created. See the discussion of deferred windows in the "Programming Structure" chapter of the *System Reference Manual*.

It should be noted that both techniques described above trade lower initial memory usage and faster load times for poorer response times when the object must be created for the first time, or when the window is made visible for the first time. Depending on the specific situation, this may or may not be the right tradeoff.

A Note on Symbols

It's worth noting that compiling with or without symbols has no effect on the in-memory size of your application. When the compiler generates symbols, they're put in a section at the end of the executable file. However, they're only brought into memory by the debugger. Thus, while stripping your program will significantly reduce the size of your application on the disk, it'll have no effect on memory usage or performance.

On the subject of compiler flags, using the **-O** (optimization) option can yield potential gains in performance, as well as somewhat smaller code, so you should always compile your production version with optimization turned on. In fact, as long as the code is also compiled with symbols, you can debug the optimized code, although be warned that the optimization sometime fool GDB.

Listings for NXWindowServerMemory() and the Timing Class

```
/*
Copyright (c) 1990 NeXT, Inc.
All Rights Reserved.

wsmemory.psw

This module defines the function NXWindowServerMemory(), which
will return the amount of Window Server memory being used by the
current Window Server context.

Synopsis:

int NXWindowServerMemory(DPSContext *context,
    int *vmUsedP, int *windowBackingP,
    NXStream *windowDumpStream);

NXWindowServerMemory() calculates the amount of Window Server
memory being used at the moment by the given Window Server
context.  If NULL is passed for the context, the current context
is used.  The amount of VM used by the current context is returned
in the int pointed to by vmUsedP; the amount of window backing
store used by windows owned by the current context is returned in
the int pointed to by windowBackingP.  The sum of these two
numbers is the amount of the Window Server's memory that this
context is responsible for.

To calculate these numbers, NXWindowServerMemory() uses the
PostScript language operators dumpwindows and vmstatus.  It takes
some time to execute; thus, calling this function in normal
operation is not recommended.

If a non-NULL value is passed in for windowDumpStream, the
information returned from the dumpwindows operator is echoed to
the NXStream given.  This can be useful for finding out more about
which windows are using up your storage.

Return Value

Normally, NXWindowServerMemory() returns 0.  If NULL is passed for
context and there's no current DPS Context, -1 is returned.
```

```
*/

#import <strings.h>
#import <stdlib.h>
#import <streams/streams.h>
#import <dpsclient/dpsclient.h>

/* These three variables define a buffer filled by _NXMyTextProc
   below, and parsed by the NXWindowServerMemory() procedure. */

static char *responseBuffer;        /* Pointer to character buffer
*/
static int   responseBufCount;  /* Number of bytes in buffer
                                   currently */
static int   responseBufSize;   /* Total size of buffer */
#define MINRESPBUFSIZE 500

/* Prototype for _NXMyTextProc; implementation is below
   NXWindowServerMemory() */

static void _NXMyTextProc(
    DPSContext context,
    char *buf,
    long unsigned int count);

/*
_NXvmstatus is a pswrap that calls the vmstatus operator after
doing a private VM garbage collection.  You need to do a garbage
collection in order to get consistent results from the vmstatus
operator.
*/

defineps static _NXvmstatus(DPSContext context|int *vmUsed)
    % First, make sure we get the status for our private VM
    currentshared false setshared
    % Do a private garbage collection
    1 vmreclaim
    % OK, get the answer
    vmstatus pop vmUsed pop
    % and restore the old shared state
    setshared
endps

int NXWindowServerMemory(DPSContext context, int *vmUsedP,
                            int *windowBackingP,
                            NXStream *windowDumpStream)
{
    int bytes,totalBytes;
```

```
DPSTextProc oldTextProc;
char *buf,*bufEnd,*newBuf;

/* First, make sure that either there is a current context or
   we were passed one. */
if (context == NULL)
{
    context = DPSGetCurrentContext();
    if (context == NULL)
        return(-1);
}

/* Reset the character buffer variables */
responseBuffer = NULL;
responseBufSize = responseBufCount = 0;

/* We'll get the Window Server VM used here.  Note that since
    the _NXvmstatus wrap function returns a result, it'll
   implicitly synchronize the given context.  This will make
   sure that all data in the buffers in either direction has
   been flushed, and that we can synchronously install our
   text proc. */

_NXvmstatus(context,vmUsedP);

/* Now install our text proc.  We're about to call the
   dumpwindows operator, which will write out a whole bunch of
   stuff on the output stream back to us.  We need to install
   our text proc before we do that, so that when the output
   comes out we will catch it.  Also, we remember the old text
   proc so we can put it back afterwards. */

oldTextProc = context->textProc;
DPSSetTextProc(context,_NXMyTextProc);

/* Now call the dumpwindows operator, and our text proc
   catches all the responses and puts them in the
   reponseBuffer. */

DPSPrintf(context,"0 currentcontext dumpwindows flush\n");

/* Call DPSWaitContext to be sure we're really done, and all
   of the responses are really in the responseBuffer. */

DPSWaitContext(context);

/* OK, now we've got all the text in responseBuffer.  We can
   take our text proc back out. */
```

```
DPSSetTextProc(context,oldTextProc);

/* The real work.  We get a line at a time out of
   reponseBuffer, and look at it to see if it looks like a
   line generated by the dumpwindows operator.  If so, we
   parse the memory usage number out of it; if not, we pass it
   to the old text proc for whatever normal processing
   occurs. */

if (responseBuffer)
{
     buf = responseBuffer;
     bufEnd = buf + responseBufCount + 1;
     *windowBackingP = 0;
     while(buf < bufEnd) {

         /* Find a line.  Null-terminate it. */

         newBuf = index(buf,'\n');
         if (!newBuf) break; /* Every line we want has a
                                newline. */
         *newBuf = 0;

         /* See if the line pointed to by buf looks like a
            dumpwindows line.  dumpwindows prints out lines
            like
              Layer: (0,448),(672,832) 129024+ bytes (131072)
            where the "+" is optional. */

     if (sscanf(buf,"Layer: (%*d,%*d),(%*d,%*d) %d%*c%*s
              (%d)", &bytes,&totalBytes) == 2)
         {   /* It's one of those lines.  Bump the storage
                count... */
             *windowBackingP += totalBytes;

             /* And echo the line to the dump stream
                (if any). */

             if (windowDumpStream)
                  NXPrintf(windowDumpStream,"%s\n",buf);
         }
         else

             /* Doesn't look like it's from dumpwindows.  Pass
                it on to the old text proc (actually, we should
                put the Return character back on before doing
                this, but that's life). */

             (*oldTextProc)(context,buf,strlen(buf));
```

```
            buf = newBuf + 1;
        } /* while(buf < bufEnd) */

        /* Free the buffer */

        free(responseBuffer);
    } /* of if (responseBuffer) */

    return(0);
} /* NXWindowServerMemory() */

/* Implementation of _NXMyTextProc.  All that _NXMyTextProc does
   is accumulate all the text that comes in on the context while
   it's active into responseBuffer, realloc'ing it when
   necessary.  It's left to NXWindowServerMemory() to parse the
   stuff out afterwards.  */

static void _NXMyTextProc(DPSContext context, char *buf,
                            long unsigned int count)
{
    /* Expand the buffer if necessary */

    while((count + responseBufCount + 1) > responseBufSize)
    {   /* Must expand buffer */
        if (responseBuffer == NULL)
        {
            responseBuffer = malloc(MINRESPBUFSIZE);
            responseBufSize = MINRESPBUFSIZE;
        }
        else
        {
            responseBuffer =
                realloc(responseBuffer,responseBufSize*2);
            responseBufSize *= 2;
        }
    }

    /* Put the new data in the buffer and update the count */

    bcopy(buf,responseBuffer+responseBufCount,count);
    responseBufCount += count;

    /* And null-terminate the buffer (this is used by the main
       procedure) */

    *(responseBuffer+responseBufCount) = 0;
} /* _NXMyTextProc */
```

```
/*
Timing.h

This class implements a simple interval timer to aid in measuring
drawing performance.  It'll measure either wall or CPU time spent
within an interval delineated by a pair of messages to the Timing
object.  It's most useful in situations where you need to measure
not only the time spent within the process, but also the time
spent in other processes, most notably the Window Server.  CPU
time includes process time, system time on behalf of the process,
and Window Server time on behalf of the process.  The results are
most accurate if averaged over a number of passes through the
interval, and the Timing object will keep track of:  number of
times entered, cumulative elapsed time, and average elapsed time.

Use the +newWithTag: method to create a Timing object.

Use the -reset message to reset the Timing object before entering
the timing interval for the first time.

A timing interval is delineated by an -enter: message and a -leave
message.  enter: takes a single argument that specifies either
WALLTIME or PSTIME.

Use the -summary: method to have the Timing object print out a
summary to the stream that is passed in as the argument to
summary:.  Alternatively, the Timing object provides methods for
querying it for the appropriate values.

Here's an example of its use.

    - action4:sender
    {
        int i=100;
        id t4 = [Timing newWithTag:4];
        [t4 reset];
        [self lockFocus];
        while(i--){
        [t4 enter:PSTIME];
        [self drawCachedLines];
        [[self window ] flushWindow];
        [t4 leave];
        }
    [self unlockFocus];
        [t4 summary:stream];
        [self addSummary];
        return self;
    }
```

```
*/

#import <objc/Object.h>
#include <sys/time.h>
#import <sys/resource.h>
#define PSTIME 0
#define WALLTIME 1

@interface Timing : Object
{
    struct timezone tzone;
    struct timeval realtime;
    struct rusage rtime;
    double synctime;
    int    stime;
    double cumWallTime; /* cum. wall time app + server */
    double cumAppTime;  /* cum. app process + system time */
    double cumPSTime;   /* cum. Server time on behalf of the
                           app */
    double avgWallTime; /* (cum. wall time app + server)/
                           cumTimesEntered */
    double avgAppTime;  /* (cum. app process + system time)/
                           cumTimesEntered */
    double avgPSTime;   /* (cum. Server time on behalf of
                           the app)/cumTimesEntered */
    double tare;        /* used to account for ipc overhead */
    int    cumTimesEntered; /* number of times timing interval
                               entered since last reset */
    int    tag;         /* identifies timer object */
    int    wallTime;    /* flag to specify whether wall or process
                           time is desired */
}

+newWithTag:(int) aTag;
    /* Creates a new timing object with tag = aTag */

-enter:(int)wt;
    /* Starts a timing interval measuring either elapsed wall time
       if wt ==WALLTIME or elapsed process time + system time +
       Server time if wt == PSTIME.  Sets the wallTime flag to be
       equal to wt. */

-wallEnter;
    /* Called by enter: if WALLTIME is desired.  You should call
       enter: rather than call this method directly. */

-psEnter;
    /* Called by enter: if PSTIME is desired.  You should call
       enter: rather than call this method directly. */
```

```
-wallLeave;
    /* Called by leave if WALLTIME was specified on the previous
       enter.  You should call leave rather than call this
       method directly.  Updates cumWallTime based on the elapsed
       time. */

-psLeave;
    /* Called by leave if PSTIME was specified on the previous
       enter.  You should call leave rather than call this method
       directly.  Updates cumPSTime and cumAppTime based on the
       elapsed time. */

-leave;
    /* Call leave to leave a timing interval.  Depending on
       whether WALLTIME or PSTIME was specified on the previous
       call to enter:, leave will invoke wallLeave or psLeave. */

-reset;
    /* Resets the values of cumWallTime, cumPSTime, cumAppTime,
       cumTimesEntered and  other variables to 0 in preparation
       for measuring a series of timing intervals.  Should be
       called prior to running a timing test. */

-avgElapsedTime;
    /* Calculates averages.  Called automatically by summary:. */

-summary:(NXStream *)c;
    /* Writes out a summary to the stream pointed to by c.
       Depending on the current value of wallTime will write out a
       summary for either wall time or ps time. */

-(double) cumWallTime;
    /* Returns cumWallTime if wallTime == WALLTIME,
       -1 otherwise. */

-(double) cumAppTime;
    /* Returns cumAppTime if wallTime == PSTIME, -1.0 otherwise.
       cumAppTime represents the cumulative time spent in the
       process and system calls made by the process.  It does not
       include time spent in the Server. */

-(double) cumPSTime;
    /* Returns cumPSTime if wallTime == PSTIME, -1.0 otherwise.
       cumPSTime represents the cumulative time spent in the
       Window Server on the behalf of the process. */

@end
```

```
/* Timing.m */

#import "Timing.h"
#import <stdio.h>
#import <streams/streams.h>
#import <dpsclient/wraps.h>
#import <appkit/graphics.h>

@implementation Timing

+newWithTag:(int) aTag
{
    self = [super new];
    tag = aTag;
    [self reset];
    return self;
}

-enter:(int)wt
{
    if(wallTime = (wt==WALLTIME))
    [self wallEnter];
    else
    [self psEnter];
    return self;
}

-wallEnter
{
    cumTimesEntered++;
    NXPing();
    gettimeofday(&realtime,&tzone);
    synctime = realtime.tv_sec + realtime.tv_usec/1000000.0;
    return self;
}

-tare
{
    struct timezone tzone1;
    struct timeval realtime1;
    struct timeval realtime2;
    NXPing();
    gettimeofday(&realtime1,&tzone1);
    NXPing();
    gettimeofday(&realtime2,&tzone1);
    tare = (-realtime1.tv_sec + realtime2.tv_sec)+
            (-realtime1.tv_usec+realtime2.tv_usec)/1000000.0;
    return self;
}
```

```
-psEnter
{
    cumTimesEntered++;
    PSusertime(&stime);
    getrusage(RUSAGE_SELF,&rtime);
    synctime = (rtime.ru_utime.tv_sec + rtime.ru_stime.tv_sec) +
                (rtime.ru_utime.tv_usec +
rtime.ru_stime.tv_usec)/1000000.0;
    return self;
}

-wallLeave
{
    double eTime;
    NXPing();
    gettimeofday(&realtime,&tzone);
    eTime = (- synctime + realtime.tv_sec +
                realtime.tv_usec/1000000.0) -tare;
    cumWallTime += eTime;
    return self;
}

-psLeave
{
    int et;
    double appTime;
    double psTime;

    getrusage(RUSAGE_SELF,&rtime);
    PSusertime(&et);
    psTime = ((et-stime)/1000.0);
    cumPSTime += psTime;
    appTime  = ((rtime.ru_utime.tv_sec + rtime.ru_stime.tv_sec) +
                (rtime.ru_utime.tv_usec +
                 rtime.ru_stime.tv_usec)/1000000.0) -synctime;
    cumAppTime += appTime;
    return self;
}

-leave
{
    if(wallTime)
    [self wallLeave];
    else
    [self psLeave];
    return self;
}
```

```
-reset
{
    cumAppTime = 0.0;
    cumPSTime = 0.0;
    cumWallTime = 0.0;
    cumTimesEntered = 0;
    return self;
}

-avgElapsedTime
{
    if(wallTime)
        avgWallTime = (cumWallTime/(double)cumTimesEntered);
    else{
        avgAppTime = (cumAppTime/(double)cumTimesEntered) ;
        avgPSTime = (cumPSTime/(double)cumTimesEntered);
    }
    return self;
}

-(double) cumWallTime
{
    if(wallTime ==WALLTIME)
    return cumWallTime;
    else
    return -1.0;
}

-(double) cumAppTime;
{
    if(wallTime ==PSTIME)
    return cumAppTime;
    else
    return -1.0;
}

-(double) cumPSTime;
{
    if(wallTime ==PSTIME)
    return cumPSTime;
    else
    return -1.0;
}
```

```
-summary:(NXStream *)c
{
    if(wallTime) {
    NXPrintf(c,"Timer %d : entered %d trials TotalWall Time  %lf
                \n",
    tag,cumTimesEntered, cumWallTime);
    }
    else {
    NXPrintf(c,"Timer %d : %d trials App: %lf  Server: %lf
                Percent Server: %lf Total: %lf\n\00",
                tag,cumTimesEntered,cumAppTime,cumPSTime,
                cumPSTime/(cumAppTime+cumPSTime),
                cumAppTime+cumPSTime);
    }
    NXFlush(c);
    return self;
}

@end
```

Document Revision Notes

This document is revised from time to time. The following lists the release date of each version of this document, and the changes from the previous version.

Date	**Changes from last version**
February 28, 1990	Created
March 26, 1990	Minor modifications in the language. No technical modifications.
April 5, 1990	Modifications in format only.