3.3 Release Notes: Application Kit

This file contains developer release notes for the 3.3, 3.2, 3.1, and 3.0 releases of the Application Kit. Items specific to or introduced in Release 3.2 are listed first, followed by the 3.1 and 3.0 notes. Some user-visible changes to the application architecture have been moved to **/NextLibrary/Documentation/NextAdmin/ReleaseNotes/AppBehavior.rtf**.

Notes Specific to Release 3.2

Known Bugs

NXRunAlertPanel

In all 3.X releases **NXRunAlertPanel** incorrectly tries to localize the arguments it receives. If one of these arguments is not found as a key in the localization string table, the system retains a reference to that pointer and uses it as the localized value. This results in the correct display, but if that argument is later freed or modified, the application gets hashtable error messages or crashes. The workaround is to be sure that you call this function with strings that you know will not change. Use either literal strings or make copies of dynamically created strings that may change later.

Text Delegate Methods

Before the Text object sends the delegate method **textWillStartReadingRichText:**, it checks to see if its delegate responds to it. Unfortunately, the text object checks to see if the delegate responds to a method named **textWillStartReadingRichText:text:runs:**; thus, the delegate method, even though it might be implemented, is never invoked. The same bug exists with **textWillFinishReadingRichText:**.

If you would like to respond to these delegate methods, implement the following stub methods in your delegate:

± **textWillFinishReadingRichText:***obj* **text:**(void *)*text* **runs:**(void *)*runs*

± **textWillStartReadingRichText:***obj* **text:**(void *)*text* **runs:**(void *)*runs*

These methods will never be invoked; so all they need to do is **return self**.

This bug exists in the 3.0 and 3.1 releases as well.

NXHelpPanel

If additional help is located in a directory other than YourProject/English.lproj/Help, and that help is to be loaded with the NXHelpPanel method, "**addSupplement:inPath:**", the help directory must not be compressed (i.e. do not invoke the "compresshelp" utility on it).

NXJournaler

Earlier versions of the documentation implied that journal recording could be made non-abortable by or'ing in the **NX_NONABORTABLEMASK** flag when calling the **setEventStatus:soundStatus:eventStream:soundfile** method. This was incorrect. In earlier versions, if this flag was included journal recording would not work. In the 3.2 version, this flag will simply be ignored if the request is for journal recording. If the request is for journal playback, the flag will be accepted and will work as it has in the past.

Notes Specific to Release 3.1

New Methods

± ignoreModifierKeysWhileDragging

This method has been added to the NXDraggingSource informal protocol. (This protocol is implemented by the object that supplies an image that can be dragged from one window to another, the source object passed as an argument in the **dragImage:**... message that initiates a dragging session.) If **ignoreModifierKeysWhileDragging** is implemented to return YES, the modifier keys held down by the user will not cause the cursor to change (to the link or copy cursor, for example), nor will they be used when deciding whether to allow the operation.

An application can use this method when implementing the dragging of private data types (e.g., patterns). The benefit is that the cursor will not switch to the copy or link cursor when the user applies the modifier keys, but the operation can still succeed and the application can apply its own semantics to the use of the modifiers. This method existed undocumented in 3.0, and is invoked under that

release as well.

This method should never return YES when the user drags a file (when the data is NXFilenamePboardType), since files can be dragged between applications. The modifier keys have well-defined meanings in the user interface for dragging files.

± (BOOL)**getMountedRemovableMedia:**(char **)*mountedMediaPathnames*
± (BOOL)**mountNewRemovableMedia:**(char **)*newlyMountedPathnames*
± (void)**checkForRemovableMedia**

These methods have been added to the NXWorkspaceRequestProtocol formal protocol, which is implemented by the object returned by the NXApp's **workspace** method. When an application is ready to let the user access files that might be on a removable disk (for example, when it puts up a custom Open panel), it can use these methods to get a list of currently mounted removable devices, and mount new devices if necessary.

getMountedRemovableMedia: fills in *mountedMediaPathnames* with a zero-terminated tab-separated list of full pathnames to all currently mounted removable media. It returns YES if it was successful in filling in *mountedMediaPathnames* and NO otherwise.

On systems that don't send an interrupt or other notification that a disk has been inserted, an application must invoke either **mountNewRemovableMedia:** or **checkForRemovableMedia** before invoking **getMountedRemovableMedia:**. Both of these methods cause the Workspace Manager to poll the drives to see if a disk is present. If a disk has been inserted but not yet been mounted, these methods will cause the Workspace Manager to mount it.
mountNewRemovableMedia: waits until the new disk has been mounted and then fills in *newlyMountedPathnames* with a zero-terminated tab-separated list of full pathnames to all newly mounted disks. It returns YES if it was successful in filling in *newlyMountedPathnames*, and NO otherwise. **checkForRemovableMedia** does not wait until the new disk is mounted; instead, it asks the Workspace Manager to mount the disk asynchronously and returns immediately.

These methods are new in 3.1, and your application will crash if you try to use them on a 3.0 system. Therefore, before you send them you should test whether the object returned by **[NXApp workspace]** responds to these methods with **respondsTo:**.

Semantic Changes

Secure Ports

In 3.0, a change was made so that, unless the user enabled the Public Window Server preference, the ports of Listener objects were registered securely (i.e., under modified names). This allowed applications launched by the user on the same machine to communicate with each other, but prevented other machines on the network from looking up their ports Because this caused problems for some existing applications that need to rendezvous over the network, the Listener class has been changed in 3.1 to register ports securely only for applications built under Release 3.0 or later. Applications built under Releases 1 or 2 will have their ports registered under the plain port name given the Listener.

Users who wish to force pre-3.0 applications to register their ports securely may set the **NXSecureAppListenerPorts** default to YES. This default may be set for particular applications or for all applications by specifying "GLOBAL" as the application name.

If a 3.0 application wishes to publish a port under a nonsecure name for intermachine use, it should override the Listener's **checkInAs:** method and publish the port under an unprotected name using **netname_check_in()**.

NXColors

Release 3.1 continues the change in direction for NXColors begun in Release 3.0.

Under 2.0, all colors were device dependent, and there were no named colors (such as PANTONE). Under 3.0 and 3.1, NXColors can basically be in one of three flavors:

- Device dependent, such as CMYK,
- Device independent, such as RGB, HSB, or grayscale, and
- Named, such as PANTONE.

Colors in the last category come with lookup tables to provide the ability to generate the correct color (CMYK, RGB, or whatever) on a given device.

This change means that colors that look the same on a device (such as the screen) might look different on another device (such as the printer). Thus, freely converting an NXColor from one type to another without regard to how the user chose it (from the color panel) might produce undesirable results on certain devices, or cause the application to discard certain aspects of the color (its device dependence or independence, its name, etc).

In line with this change in direction, Release 3.1 makes these specific modification:

- In 3.1, **NXEqualColor()** will return YES if and only if the two colors will produce the same results on any printer. This means that a color created from CMYK components is not equal to a color created from RGB components, as the

former is device dependent and the latter is device independent. Similarly, a PANTONE color is never equal to a color created from CMYK or RGB components.

- Under 3.0, functions such as **NXConvertColorToCMYK()** and **NXYellowComponent()** reported the CMYK components of PANTONE colors by simply converting the screen color (specified in RGB) to CMYK. The results were not satisfying to users, as they lacked the K component and the CMY components were meaningless.

  Under 3.1, when an NXColor is asked for its CMYK color components (if it doesn't record them directly), it will first attempt to get or compute them before resorting to the simple-minded conversion between its own color components and the desired ones. This means that if a PANTONE (or other named) color is asked for its CMYK components, the values will be obtained from the DefaultCMYK device table. This feature can be disabled by setting the value of the **NXSmartNamedColorConversion** default to NO.

Under 3.1, developers may wish to test their applications with the **NXCMYKAdjust** default enabled (as described above), and assure that the application behaves reasonably both on the screen and on the printer. This will help verify that the application uses NXColors and the Color panel correctly. Note that when CMYK adjustment is enabled, the algorithms that convert between RGB and CMYK

components of a color also change; they return values that match the way the colors appear on the screen. This could be an area of trouble in some applications.

Other Changes

New Utility

A new utility called **compresshelp** has been released in the **/usr/bin** directory. If you are using the NeXT Help System, and have one or more help directories in your project, this utility can be used to compress the files in those directories into one file called *DirectoryName***.store**. In the case of the default Help directory (named "Help"), a file called **Help.store** will be created. When Help is invoked for your application, the Help system will first look for a **Help** directory and then for a **Help.store** file. By default a **Help.store** file, and not a **Help** directory will be placed in your ***.app**.

The **.store** file created by **compresshelp** will contain **only** files found in the **Help** directory which have a **.rtf**, **.rtfd**, **.tiff**, or **.eps** extension. **compresshelp** will warn you about files it has skipped.

By default **compresshelp** will create a compressed output file named

*DirectoryName***.store**. However if the "-o OutputFileName" option is specified, the compressed help will be placed in a file named *OutputFileName***.store.**

Typically you will not need to invoke this utility directly, as it is invoked automatically by the Makefiles supplied by Project Builder for default help directory, named **Help**. If you have other help directories, and you want them to be compressed, you will have to explicitly invoke this utility in your Makefile.postamble.

Trapping Illegal Floating Point Operations

A new default named **NXTrapIllegalFloatingPointOps** has been added to allow catching illegal IEEE floating point operations. These operations include dividing 0 by 0, invalid comparisons to NaN, and converting out-of-range floating point numbers to integral values. Running with this default set to YES enables the trap:

```
appname.app/appname -NXTrapIllegalFloatingPointOps YES
```

If your application performs an illegal operation, a floating exception will be raised and the application will crash, allowing you to debug the problem. Given that the results of most invalid floating point operations are machine-dependent, using this default may help you chase down floating-point related problems encountered while porting between architectures. Don't leave this default enabled except when debugging.

Bug Fixes

The following bugs are among those that have been fixed in the 3.1 version of the Application Kit:

Developer Bugs

Reference	20657
Problem	When the user changed the update mode of a data link to "continuous," an out-of-date link would not immediately update.
Description	Users expect "continuous" data links to be "in sync" with the source data. In 3.1, when the update mode is changed to "continuous," the system immediately checks whether the data is current, and if not, updates it.

Reference	29821
Problem	Filter services weren't executed with the full path.
Description	The Application Kit invoked filter services using only the application name, not the full pathname. Some filter services expected the full

path name. In 3.1, the Kit executes all filter services with the full pathname.

Reference 29866

Problem Data links with a "continuous" update mode would not update when changes made to the source document were reverted.

Description A "continuous" data link should ask the source document for new data when changes made to the document are undone. In Release 3.1, it does.

Reference 30009

Problem The frames that outline linked data in response to the Show Links command were not drawn correctly in scaled coordinates.

Description Corrected in 3.1.

Reference 30234

Problem **fileOperationCompleted:** methods were not being invoked.

Description When requested to perform a file operation, the Workspace Manager might choose to do it asynchronously. If so, the Application object's delegate is supposed to receive a **fileOperationCompleted:** message notifying it that the operation was accomplished. In 3.0, the message wasn't getting through.

Reference 30393

Problem Some NXCachedImageRep objects would generate an error when archived.

Description The **NXSizeBitmap()** function was incorrectly calculating the size of some images.

Reference 30694

Problem The data links mechanism strips the **/private** prefix on pathnames, sometimes resulting in invalid paths.

Description Removing **/private** from a pathname sometimes produces a bogus path, typically where the directory is **/private/tmp**. In 3.1, **/private** is stripped only where the result is a valid and equivalent path.

Reference 30807

Problem It wasn't possible to read images from more than one Display PostScript context.

Description After reading (getting bitmap data for) a rendered imageÐfor example, by invoking NXBitmapImageRep's **initData:fromRect:** methodÐin one context, an application could not then read an image in a different context. The problem has been corrected.

Reference 31024

Problem **draggedImage:beganAt:** notifications were received before the dragged image was on-screen.

Description In 3.0, there was a race condition between **draggedImage:beganAt:** messages and the dragged image appearing on-screen. This race made it difficult for **draggedImage:beganAt:** methods to erase underneath the dragged image without an annoying flash. In 3.1, you are guaranteed that the dragged image is on-screen before the message is sent, and thus you can erase under the image without flashing.

Reference	31049
Problem	NXDataLink's **initLinkedToSource:**... method would, on rare occasions, cause a crash.
Description	The cause of the crash has been removed.

Reference	32870
Problem	windowWillMove: sender is not the Window.
Description	In 3.0, the window delegate message **`windowWillMove:`** would pass a view from the Window as the sender parameter instead of the Window itself. It has been corrected in 3.1 to pass the window. Apps compiled under 3.0 that worked around this bug will not be affected..

Notes Specific to Release 3.0

These notes were included with the Release 3.0 version of the Application Kit. Sections that are no longer relevant have been marked with an italicized comment.

Known Problems

Known problems in the Application Kit:

- The command line program **`genstrings`** will crash if the specified output file cannot be created.

Incompatible Changes Since 2.1

Incompatible changes made to the Application Kit since Release 2.1:

- The global variable **`NXSelectionPboard`** has been renamed to **`NXGeneralPboard`**.

- The following Speaker/Listener messages have been obsoleted and replaced with improved API described below under "Interapplication Image Dragging" and "Workspace Protocol".

```
- (int)registerWindow:(int)windowNum toPort:(port_t)aPort;
- (int)unregisterWindow:(int)windowNum;
```

```
- (int)icon¼; (the set of messages received during dragging)
- (int)launchProgram:(const char *)name ok:(int *)flag;
- (int)getFileInfoFor:(char *)fullPath app:(char **)appname
    type:(char **)type ilk:(int *)ilk ok:(int *)flag;
- (int)getFileIconFor:(char *)fullPath TIFF:(char **)tiff
    TIFFLength:(int *)length ok:(int *)flag;
- (int)unmounting:(const char *)fullPath ok:(int *)flag;
- (int)powerOffIn:(int)ms andSave:(int)aFlag;
- (int)extendPowerOffBy:(int)requestedMs actual:(int *)actualMs;
```

- The function **PSsetpattern()** no longer takes an argument. This change is only visible to programs recompiled under 3.0, and will not affect existing applications built for a 2.x release.

- The constants **NX_RESIZEBUTTONMASK** and **NX_ALLBUTTONS** have been removed for 3.0. In place of using **NX_RESIZEBUTTONMASK**, you should create a window with style **NX_RESIZEBARSTYLE**. In place of **NX_ALLBUTTONS** you should use (**NX_CLOSEBUTTONMASK|NX_MINIATURIZEBUTTONMASK**).

- **PrintInfo** has been changed to use the **NXPrinter** object for specific information about the printer that is being used. See the description of **NXPrinter** above. Thus, the following methods have been removed from the public API:

```
- setManualFeed:(BOOL)flag;
- (BOOL)isManualFeed;
- setPrinterName:(const char *)aString;
- (const char *)printerName;
- setPrinterType:(const char *)aString;
- (const char *)printerType;
- setPrinterHost:(const char *)aString;
- (const char *)printerHost;
- setResolution:(int)anInt;
- (int)resolution;
```

- Because of the addition of **`updateFromPrintInfo`** and **`finalWritePrintInfo`** (see below), the following methods have been removed from PrintPanel API:

```
- readPrintInfo;
- writePrintInfo;
```

- The following methods have been removed from the NXColorPanel API :

```
+ new
+ newColorMask:
+ newContent:style:backing:buttonMask:defer:
+ newContent:style:backing:buttonMask:defer:colorMask:
- updateCustomColorList
- (int)colorMask;
- setColorMask:(int)colorMask
```

The only initializer for this class is now

```
+ sharedInstance:(BOOL)create
```

The **`setColorMask:`** method was always a no-op. The initial color mask can now only be set before the ColorPanel is instantiated, using the **`setPickerMask:`** method (see below).

- The following constants have been renamed in the NXColorPanel API :

```
NX_CUSTOMCOLORMODE            ->      NX_COLORLISTMODE
NX_BEGINMODE                  ->      NX_WHEELMODE

NX_CUSTOMCOLORMODEMASK        ->      NX_COLORLISTMODEMASK
NX_BEGINMODEMASK              ->      NX_WHEELMODEMASK
```

- **`NXReadBitmap()`**, **`NXSizeBitmap()`**, and **`NXDrawBitmap()`** have been moved from `tiff.h` to `graphics.h`. Because `tiff.h` imports `graphics.h`, this shouldn't cause any source changes. **`NXImageBitmap()`**, now obsolete, has been removed from `tiff.h`.

- In 2.1, in windows that become key only if needed a view that did not accept first

mouse would be passed the mouse clicks even if the window was not key. In 3.0 this has been fixed. Custom controls in these windows should return **YES** to **acceptsFirstMouse:**. Applications compiled under 2.1 are not affected by this change.

- The global variables **NXSystemFont** and **NXBoldSystemFont** have been removed from the API. Use the new font methods below in their place.

- The location field in flags changed events is no longer valid in 3.0. Also note that the location field in key down and key up events has never been valid.

- The defaults.h header file has moved from the appkit headers directory to the defaults directory. Thus imports of **<appkit/defaults.h>** should be changed to **<defaults/defaults.h>**

New Features

- Color dragging

  Color-dragging is now accomplished using the new dragging API (see below). Thus, though supported, the **acceptColor:atPoint:** method should be avoided. For those who continue to use **acceptColor:atPoint:** in any case, two points should be kept in mind: 1) This method will now be properly called on

flipped views (this did not used to be the case in earlier versions) and 2) If your view implements both the dragging protocol *and* the **`acceptColor:atPoint:`** method, **your `acceptColor:atPoint:` method will never get called!!**

To support the new color-dragging methodology, a new Pasteboard type has been added called **`NXColorPboardType`**. You can read and write colors from the Pasteboard using the following functions:

```
NXColor NXReadColorFromPasteboard(id pasteboard);
void NXWriteColorToPasteboard(id pasteboard, NXColor color);
```

Here is a look at what your code might look like to replace your current **`acceptColor:atPoint:`**

```
- initFrame:(NXRect const *)theFrame
{
   [super initFrame:theFrame];
   [self registerForDraggedTypes:&NXColorPboardType count:1];
   return self;
}

- (NXDragOperation)draggingEntered:(id <NXDraggingInfo>)sender
{
   if ([sender draggingSourceOperationMask] &
     NX_DragOperationGeneric) {
```

```
      return NX_DragOperationGeneric;
    } else {
      return NX_DragOperationNone;
    }
}

- (BOOL)performDragOperation:(id <NXDraggingInfo>)sender
{
    NXPoint p = [sender draggingLocation];
    NXColor c = NXReadColorFromPasteboard(
                                [sender draggingPasteboard]);
    [self acceptColor:c atPoint:&p];
    return YES;
}
```

- Application Tile

  A new named image, **`NXAppTile`**, has been added to allow apps to draw in their application icon without destroying the standard look. Developers who want to draw in their application icons should find this image with **`findImageNamed:`** and composite it at the lower left corner of the content view of the window. Then any custom drawing should be done above, *centered* on this image.

- RTFD support in the Text object

There is now support in the text object for reading, writing, and editing `.rtfd` file format.

```
- setGraphicsImportEnabled:(BOOL)flag;
```

Enables dragging of graphics into the text object.

```
- (BOOL)isGraphicsImportEnabled;
```

Returns whether graphics importing is enabled.

```
- (NXRTFDError)saveRTFDTo:(const char *)path
            removeBackup:(BOOL)removeBackup
            errorHandler:errorHandler;
```

Saves the contents of the text object to an `.rtfd` format file. The file name passed in should end with `.rtfd`. If *removeBackup* is **YES**, the backup document that is created to safely save the document is removed.

```
- (NXRTFDError)openRTFDFrom:(const char *)path;
```

Loads an `.rtfd` document into the text object.

```
- writeRTFDTo:(NXStream *)stream;
```

Writes a flattened form of the `.rtfd` document into a stream. See **`readRTFDFrom:`** to read the contents of such a stream.

```
- readRTFDFrom:(NXStream *)stream;
```

Reads from a stream an `.rtfd` document, written out by the **`writeRTFDTo:`** method.

```
- writeRTFDSelectionTo:(NXStream *)stream;
```

Writes a flattened form of the `.rtfd` selection into a stream. See **`replaceSelWithRTFD:`** to read the contents of such a stream.

```
- replaceSelWithRTFD:(NXStream *)stream;
```

Reads from a stream an `.rtfd` document, replacing the current selection. This reads the output of the **`writeRTFDSelectionTo:`** method.

- New Filter Services Support

There are three new types of services: Filter services, Print Filter services and

Spell-Checking services.

Filters are services which have no Services menu entry and which have the semantic of converting a piece of data from one type to another. Filter services are accessed via the following Pasteboard methods:

```
+ (NXAtom *)typesFilterableTo:(const char *)type;
+ newByFilteringFile:(const char *)filename;
+ newByFilteringData:(id <NXData>)data
              ofType:(const char *)type;
+ newByFilteringTypesInPasteboard:(Pasteboard *)pboard;
```

The last three create a Pasteboard which has declared in it all the types which the argument to the method can be filtered to. When you actually ask for the data of a type in the returned Pasteboard, the filter service will be invoked. Thus, calling these methods is reasonably cheap until you actually ask for the data.

The first method is sort of the opposite of the last three. Given a type, it will return a null terminated list of types which can be converted TO that type.

For convenience (since many filter services operate on files), the following functions are provided to go between filename/contents pasteboard types and file extensions:

```
NXAtom NXCreateFilenamePboardType(const char *fileType);
NXAtom NXCreateFileContentsPboardType(const char *fileType);
const char *NXGetFileType(const char *pboardType);
const char **NXGetFileTypes(const char *const *pboardTypes);
```

Creating a filter service is very similar to creating a regular service except that instead of using the "Message:" keyword in your Services description, you use the "Filter:" keyword.

In addition, there are a few special kinds of "Port:" you can use for filter services...

For those which operate on **NXFilenamePboardType**, there are special ports called **NXUNIXSTDIO** and **NXMAPFILE**. The first will cause your filter to get launched with the name of the file to be converted on the command line and which expects the data to be put in the Pasteboard to come out of the filter's stdout. This is mostly for compatibility with existing filters which work this way, it is strongly encouraged that you NOT use this mechanism because it causes the filter to be repeatedly launched rather than staying around and waiting on a Listener port (which is much more efficient). The second one is for file types which map directly to primitive pasteboard types. An example of this is **.rtf** files which map directly to the **NXRTFPboardType**.

The other special "Port:" is for associating types which can be converted from one to the other with no actual modification to the data. It is called **NXIDENTITY**. An

example of this is the **.eps** file format which can be "filtered" to the **.ps** format with no change. Remember to be clear about which way the filter is filtering when you use this port (because, for example, a **.ps** file can NOT be filtered to **.eps** using this special port, only the other way around).

Print Filter services are a specialized form of services which are largely unrelated to Filter services. Essentially, these filters take PostScript on their stdin and munge it and write the result out to a file specified on their command line via "-o outputfile.ps". Print Filters can also generate non-.ps files (but still from PostScript input). A services description for a Print Filter looks like this:

```
Print Filter: superps
Executable: ps2superps
Menu Item: Super PostScript Output
Device Dependent: YES
```

The *superps* is the file extension to use and is optional and, if not present, defaults to "ps". The Executable is the name of the filter (just like in any other Service not provided by an application). The Menu Item will appear in a PopUpList in the SavePanel brought up by the PrintPanel when you choose Save in the PrintPanel. Device Dependent is optional (and defaults to NO)--it tells the printing machinery whether to generate PostScript (which will be stdin to the Print Filter) which is specific to the type of printer the user has chosen in the PrintPanel before clicking the Save button. The Menu Item is fully localizable through the Services

localization mechanisms.

Spell-checking services are accessible via the new NXSpellServer and NXSpellChecker classes (see under **New Classes**).

- New Pasteboard Type

In order to better support Filter services and other processes which want to rendezvous on communicating files as a data type, a new Pasteboard type has been added to the AppKit:

```
NXFileContentsPboardType
```

The first type can only be written or read from the Pasteboard using the methods:

```
- (BOOL)writeFileContents:(const char *)filename;
- (char *)readFileContentsType:(const char *)type
                        toFile:(const char *)filename;
```

The format of the data written/read to/from the Pasteboard by these methods is unspecified in 3.0. These methods are analogous to the methods **`writeType:...`**, so you must be sure to declare **`NXFileContentsPboardType`** before calling the **`writeFileContents:`** method.

- TIFF 6.0 Support

  The TIFF library has been updated to support revision 6.0 of TIFF. It should still be the case that all TIFFs that could be read with the earlier versions of NEXTSTEP can still be read, and, as an added bonus, some new images can also be understood.

  Under NEXTSTEP 2.0 JPEG images were written using a very early draft of the new TIFF standard. Because of numerous revisions to this spec, JPEG images written under 2.0 are pretty much *not* compatible with anything, except that they can be loaded under NEXTSTEP 3.0. JPEG images written under 3.0 are compatible with the TIFF 6.0 draft but cannot be read back on earlier versions of NEXTSTEP.

  Note that because JPEG compression can be applied to 8-bit images only, non-8-bit images will be written out as 8-bit when compressed using JPEG.

  The TIFF library now also allows writing one-bit images with CCITT Group 3 and Group 4 compression. It is an error to try to write deeper images with these compression types.

  One of the other changes in NEXTSTEP 3.0 is the way the presence of alpha is indicated in the TIFF file. Under 2.0, the private **`Matte`** tag (32995) was used to

indicate that the extra channel of data was to be interpreted as alpha and the image data was premultiplied. 3.0 uses the TIFF 6.0-blessed **ExtraSamples** tag (338) to accomplish the same thing. Because of the safety checks built into the TIFF reader, 2.0 is actually able to read 3.0 files and deduce that the extra samples are alpha, despite the fact that the **Matte** tag is no longer written out. Thus files with alpha are interchangable between 2.0 and 3.0 systems. However, tiffutil under 2.0 is not smart enough to indicate that the file has alpha when the **-info** option is used; thus some of the output from tiffutil on a 2.0 system will be misleading on alpha images written under 3.0. The **-dump** option will indicate the presence of unrecognized tag 338 with a value of 1.

- Dynamic Services

  You can now dynamically add Services menu entries to the system by creating a file or directory with the extension **.service** and placing it in your normal application path or one of **/NextLibrary/Services**, **/LocalLibrary/Services** or **~/Library/Services**. The contents of this file (or, if a directory, a file called "services" inside it) is exactly equivalent to the contents of a services control file. After adding the file, call the function **NXUpdateDynamicServices(void)** to get the system to recognize your newly-added services.

- Interapplication Image Dragging (declared in **<appkit/drag.h>**,

<appkit/View.h> and **<appkit/Window.h>**)

New support has been added to View and Window for dragging images within and between applications. Since the actual data transfer is pasteboard based, any type of data may be transfered (not just filenames). A View that wishes to be the destination for dragged images sends itself the following message, which registers the Pasteboard data types that it is willing to accept. (Note that to receive files, use the **NXFilenamePboardType** type, *not* extensions like ".tiff" or ".eps").

```
- registerForDraggedTypes:(const char *const *)newTypes
                    count:(int)numTypes
```

When an image is dragged into and out of the View, the View is sent messages so it can respond to the dragging interaction. These methods are all passed a ***sender***, which can be queried for information about the object being dragged in.

```
- (NXDragOperation)draggingEntered:sender
```

... is sent to the view when the dragged image first enters the view. The receiver returns a `NXDragOperation` to indicate which operation on the data it will perform (move, copy, link), which is used to set the cursor as the user drags the image. The view should first ask ***sender*** for drag source's operation mask (see below) to know what operations it can choose from. The default behavior is to return

NX_DragOperationNone, which indicates the dragged image will not be accepted.

- (NXDragOperation)draggingUpdated:*sender*

... is sent periodically while the image being dragged remains within the View. As with **draggingEntered:**, an **NXDragOpMask** is returned. The default behavior is to simply return the value last returned by **draggingEntered:**.

- draggingExited:sender

... is sent when the image is dragged out of the view.

- (BOOL)prepareForDragOperation:*sender*

... is sent when the image is dropped on the view. It returns whether the View will accept the data represented by the dragged image. The default behavior is to return **YES**.

- (BOOL)performDragOperation:*sender*

... is sent after the dragged image is removed from the screen. The View should implement this method to do the real work of importing the data represented by the image. It returns whether the View accepted the data. The default behavior is to

return **NO**.

```
- concludeDragOperation:sender
```

... is sent after the complete dragging interaction is finished, if the destination accepted the data.

From within the above methods, the receiver can send ***sender*** a number of messages to find out about the drag session:

`- draggingDestinationWindow;`	The Window receiving the image
`- (NXDragOperation)draggingSourceOperationMask;`	The operation mask supplied by the source, anded with the operation indicated by the user by holding down any modifier keys.
`- (NXPoint)draggingLocation;`	The current location of the mouse in the Window's coordinate system
`- (NXPoint)draggedImageLocation;`	The current location of the dragged image in the Window's coordinate system
`- (NXImage *)draggedImage;`	An **NXImage** of the dragged image
`- (NXImage *)draggedImageCopy;`	A copy of an **NXImage** of the dragged

	image
`- (Pasteboard *)draggingPasteboard;`	A **`Pasteboard`** with the data represented by the dragged image
`- (BOOL)isDraggingSourceLocal;`	Whether the source is in the same application
`- draggingSource;`	The source object (if dragging source is local)
`- (int)draggingSequenceNumber;`	An int identifying this image dragging session

The object returned by **`draggedImage`** remains valid for the duration of the drag interaction. The object returned by **`draggedImageCopy`** can be retained indefinitely by the caller, and will not be freed automatically. Use **`draggedImage`** when you just need to give feedback during the drag interaction. Use **`draggedImageCopy`** when you need to retain the image for use after the drag is completed. Usually you won't call **`draggedImageCopy`** until you know the image has been deposited on your View or Window.

The following method can be sent (only from **`prepareForDragOperation:`**) to the sender to cause the icon to slide to a certain location after the dragged image has been released and accepted:

```
- slideDraggedImageTo:(NXPoint *)screenPoint;
```

All of the above methods can also be used at the Window level instead with a particular View. **`registerForDraggedTypes:count:`** is sent to the Window (instead of to a View), and the Window's delegate receives the messages when an images enters, exits, etc.

To be the source of a drag, an application initiates the dragging session by sending the following message to a View or Window:

```
- dragImage:anImage
          at:(NXPoint *)location
      offset:(NXPoint *)initialOffset
       event:(NXEvent *)mouseDownEvent
  pasteboard:(Pasteboard *)dragPasteboard
      source:sourceObj
   slideBack:(BOOL)slideFlag;
```

`anImage` is the image to be dragged. *`location`* is initial location of the image. *`initialOffset`* is the amount the mouse has moved since the initial mouse down event. This should be (0,0) unless you track the mouse before calling this method (possibly to implement some hysteresis before initiating the drag session). A NULL pointer is interpretted as (0,0). *`mouseDownEvent`* is the original mouse down event that was received to start the drag session. *`dragPasteboard`* is a

pasteboard containing the data to be transfered to the destination. The source should aquire this pasteboard by using `[Pasteboard newName:NXDragPboard]`, and declare the data types it can supply to that pasteboard before calling **`dragImage:`**... . *`sourceObj`* is an object that will receive messages described below during the drag sesssion. *`slideFlag`* controls whether the image will slide back to the source if it is not successfully accepted by some destination.

During the drag session, the *`sourceObj`* is sent the following messages:

```
- (NXDragOperation)draggingSourceOperationMaskForLocal:(BOOL)flag
```

... is sent to request the mask of dragging operations available. *`flag`* indicates whether the potential destination object is in the same application.

```
- draggedImage:(NXImage *)image beganAt:(NXPoint *)screenPoint
```

... is sent after the dragged image has been displayed on the screen, but before it has begun following the mouse.

```
- draggedImage:(NXImage *)image endedAt:(NXPoint *)screenPoint
    deposited:(BOOL)flag
```

... is sent after the dragged image has been released. *flag* indicates whether the image was successfully deposited on a destination.

Using the NXDragOperation's: The drag operation provided by the source and destination are used to help the apps negotiate the type of operation that will be performed, and to give the user feedback via the cursor shape. The possible values are:

```
NX_DragOperationNone
NX_DragOperationCopy
NX_DragOperationLink
NX_DragOperationGeneric
NX_DragOperationPrivate
```

"None" means no operation will be taken. "Copy" means that the data being dragged will be copied when it is deposited. "Link" means a link of some sort will be made to the source of the data. "Generic" means the data being dragged will be accepted and the "standard" operation for the given scenario will be performed. For example, in the "Generic" case the Workspace Manager will move files. In color dragging "Generic" simply means a successful association of the dragged color with the destination. If data is not being clearly copied or linked *in the user's model* "Generic" is the right operation to use. "Private" means special messages will be exchanged between the source and destination. In this case, the system does not alter the cursor appearance.

A source should respond to **draggingSourceOperationMaskForLocal:** with a *mask* of as many operations as it supports. This should almost always include `NX_DragOperationGeneric`, and `NX_DragOperationCopy` if this is applicable (it usually is for files). `NX_DragOperationLink` should be included only if there is an expected linking mechanism for the data, i.e., a destination may make an ObjectLink to a file. You may not want to include `NX_DragOperationLink` if the source is a file inside a file package, since it may be dangerous for the user to directly edit this data without the container's knowledge.

A destination responds to its messages with a *single operation*, not a mask. A destination should always ask for the **draggingSourceOperationMask** and choose an operation supplied by the source, or else return `NX_DragOperationNone` if no operation is workable. You are also free to examine the available data types, if this affects your decision. If the user is using no modifier keys, the entire mask provided by the source will be visible, and you should choose the best default operation (usually "Copy" for moving files, "Generic" for other types, rarely "Links"). If the user holds down a modifer key, his preference will mask the operations supplied by the source. (The Control key is "Link", Alternate is "Copy", Command is "Generic"). In this case, the destination will see only one available operation. If it is supported, the destination should return that operation, else `NX_DragOperationNone`.

- New Defaults

  The default **NXUseTrueGrays** has been added. If set to **YES**, then areas of the UI that use patterns to implement various gray dithers will instead be drawn with the real gray value. The most noticeable example of this is in Scrollers. This default can be useful when grabbing bits for screen shots.

  The defaults **NXSystemFonts** and **NXBoldSystemFonts** have been added for settting the system font and bold system fonts of an application or the system. Their values are a list of semi-colon separated font names. The first available font is used.

  The **GLOBAL** default **NXLanguages** has been added. It can be used to set the prefered languages to use on an application by application basis. Its value is a list of languages separated by semi-colons. Apps should use Application's **systemLanguages** method intead of reading this default directly.

  The application defaults **NXUseCalibratedColor** and **NXColorCalibrateLevelOneOps** have been added to disable color calibration features of the kit for applications that break or exhibit strange behaviour. These two defaults are described elsewhere in these notes.

The application default **NXMaxSharedImageWidth** has been added to control how non-unique NXImages share caches within an application. This default should be used for debugging purposes only and is described elsewhere in these notes.

New Classes

- **NXDataLinkManager, NXDataLink, NXSelection, NXDataLinkPanel**

  These four classes have been added to the AppKit to support the dynamic linking of data between documents of different applications. Information on this feature can be found in the ObjectLinks release notes.

- **NXColorList**

  Instances of NXColorList manage lists of named colors. Example of color lists include PANTONE and the various lists that appear in the "Custom Color" mode of the Color Panel. In addition, an application can use NXColorLists in managing document-specific color lists.

  Color lists come in two varieties: Ones that generate *named* colors and ones that don't. By default, color lists generate unnamed colors; thus, once the color is handed out, it has no idea what its name was or what list it came from.

Named colors not only remember their color value but also the name of the list and the color. When they are asked to print the color (through **`NXSetColor()`**), these colors generate a reference to the color list and name to allow the printing system to search for the correct value for the device. Only lists such as PANTONE are set up to generate named colors. Such lists are also considered immutable and attempts to change them at runtime will raise errors.

Color lists, except for those created at runtime and saved out programmatically, are stored in file-wrappers. This allows for localized and/or device-dependent versions of the lists.

- **`NXColorPicker`**

  This class is added only for subclassing, and can be used by application developers who want to "inject" their own color pickers into the ColorPanel. This class conforms to the NXColorPicking protocol.

- **`NXSpellChecker`**

  It is now possible to access the NeXT spell checking mechanism from your application. In other words, the NeXT spell checker can spell check other objects besides our Text object.

For an object to be "spell-checkable", it need only implement the following two protocols:

NXReadOnlyTextStream

```
- openTextStream;
- (BOOL)seekToCharacterAt:(int)offset relativeTo:(int)seekMode;
- (int)readCharacters:(char *)buffer count:(int)count;
- (int)currentCharacterOffset;
- (BOOL)isAtEOTS;
- closeTextStream;
```

NXSelectRange

```
- (void)selectCharactersFrom:(int)start to:(int)end;
- (int)selectionCharacterCount;
- (int)readCharactersFromSelection:(char *)buffer
                             count:(int)count;
- (void)makeSelectionVisible;
```

The first is used by spell-checking services to suck the text out of the object you want spell-checked. The characters should be NEXTSTEP encoding characters (so, if you run across some Kanji text or other text whose characters do not exist anywhere in the NEXTSTEP encoding, please turn those characters into a "space" character).

Note that the object may choose to break itself up into lots of little **NXReadOnlyTextStream**'s. An example of this is if you can mark ranges of text with a language in your application, you would probably want to return YES to **`isAtEOTS`** (and stop returning characters via **`readCharacters:count:`**) whenever you get to the end of a string of text in a single language. Then, you can tell the NXSpellChecker object to switch languages and start up another spell check.

Note that the **`currentCharacterOffset`** should always reflect either the position of the insertion point of the text, or the start of the selection in the text, or, if neither of those exist, the start of the text. Note that this means that the **NXReadOnlyTextStream** is only expected to be valid during the duration of a call to **`checkSpelling:of:`**.

A spell-check is initiated by sending ...

```
- (BOOL)checkSpelling:(NXSpellCheckType)type
    of:(id <NXReadOnlyTextStream, NXSelectRange>)textStream;
```

... to the NXSpellChecker's **`sharedInstance`**. The spell-check ends either when a misspelled word is found or when the NXReadOnlyStream runs out of characters. It returns whether a misspelled word was found.

type almost always wants to be **NX_CheckSpelling**, in which case, if the spell-check was initiated at a **currentCharacterOffset** not equal to 0 and a misspelled word is not found by the time the **EOTS** is reached, the NXSpellChecker will attempt to wrap around to the beginning of the text and continue spell-checking (until it reaches the original start point).

Calling **checkSpelling:of:** with a *type* of **NX_CheckSpellingToEnd** will not wrap around to the start of the text. **NX_CheckSpellingFromStart** starts the spell checking at the beginning of the NXReadOnlyTextStream. There are other modes as well, see the header file.

The spell-checker also allows you to insert an accessory view into the spell-checking panel.

You can control the language of spell-checking programatically by calling **setLanguage:**. The arguments to this function are the same as those returned by **systemLanguages** (they are strings registered by NeXT for each language, please check with NeXT to determine what they are or register new ones). Be careful to give the user some control over this, because calling this method will blast whatever choice the user may have made in the spelling panel. You should probably only do this if the user has explicitly set the language he wants in the document somehow, and, even then, let him turn that auto-language-setting off.

If you want your object to support correction of misspellings, it should implement the NXChangeSpelling protocol which only has one method, **`changeSpelling:`**, whose argument is a Control which the receiver should ask the **`stringValue`** of to and replace its selection with whatever that **`stringValue`** returns.

If you want your document to have its own "ignored word" list, your textStream must implement the **NXIgnoreMisspelledWords** protocol which has only one method:

```
- (int)spellClientTag;
```

This should return a unique integer.

If you implement this method, then the Ignore button in the spelling panel will be enabled and you are eligible to use the following methods:

```
- (char **)ignoredWordsForSpellClient:(int)tag;
- setIgnoredWords:(const char *const *)words
    forSpellClient:(int)tag;
- closeSpellClient:(int)tag;
```

The first two are for getting and setting the ignored word list. You can use these to store the ignored word list with your document and reset it up when the document

is opened again. You should call the last one when your document is closed so that the storage of the ignored words is freed up.

Finally, you may want to make it easy on your user by putting whatever word he selects in your document into the NXSpellChecker's word field. You should probably only do this on double-click (i.e. only do it when the user actually selects a word). This is accomplished via ...

```
- setWordFieldValue:(const char *)aWord;
```

- **NXSpellServer**

It is now possible to write your own spelling checker for the NeXT machine. You do this simply by implementing an object which knows how to do two things: find a misspelled word in a stream of text and suggest guess for a misspelled word.

A spell server is never an application, it is a simple program whose **main()** looks something like this ...

```
void main()
{
   NXSpellServer *server = [[NXSpellServer alloc] init];
   if ([server registerLanguage:"English" byVendor:"ACME"]) {
      [server setDelegate:[ACMEEnglishSpellChecker new]];
```

```
        [server run];
        fprintf(stderr, "Unexpected death!\n");
    } else {
        fprintf(stderr, "Unable to check spell-checker in.\n");
    }
}
```

... where the **`ACMEEnglishSpellChecker`** is a simple subclass of **Object** which implements the following two delegate methods:

```
- (BOOL)spellServer:(NXSpellServer *)sender
 findMisspelledWord:(int *)start
             length:(int *)length
         inLanguage:(const char *)language
       inTextStream:(id <NXReadOnlyTextStream>)textStream
         startingAt:(int)startPosition;

  - (void)spellServer:(NXSpellServer *)sender
suggestGuessesForWord:(const char *)word
           inLanguage:(const char *)language;
```

A single server can spell-check in many languages (i.e. repeated calls to **`registerLanguage:byVendor:`** is perfectly acceptable).

See the description of the **NXReadOnlyTextStream** above (under NXSpellChecker) to see how your spell-checker can suck text out of the object

being spell-checked. The return value for **findMisspelledWord:** is whether or not a misspelled word was found. If one was found, then **start** (and **length**) should be set to the start (and length) of the misspellled word in the **textStream**.

The implementation of **suggestGuessesForWord:** should call the NXSpellServer method:

```
- addGuess:(const char *)aWord;
```

whenever it figures out another guess for the passed-in ***word***.

The system also maintains a simple user dictionary which you can check the contents of via:

```
- (BOOL)isInUserDictionary:(const char *)word
              caseSensitive:(BOOL)flag
```

This method is very efficient and appropriate for calling inside inner loops, in fact, you may want to use the Objective-C **methodFor:** mechanism to get a pointer to a C function which implements this method.

- **NXHelpPanel**

There is now a class **NXHelpPanel** for applications to use when presenting help to the user. It is simple to use and there is very little API for the programmer to be aware of . To enable the help panel to be shown you will need to put a "Help..." menu item in your Info submenu. It should have the First Responder (NULL) as its target and **showHelpPanel:** as its action. This item will display the help panel. Initially the help text is set to the first item in the table of contents of your **Help** directory. Subsequent times the help panel is displayed it shows whatever was last shown when the panel was hidden (however, if you are displaying context-sensitive help, see below). If you get your Info menu from InterfaceBuilder it will already have a "Help..." item correctly set for you.

The main task for the developer is to create and customize a **Help** directory in the various .lproj directories in your application wrapper. The preferred way create a Help directory is to have ProjectBuilder copy a template into the .lproj for you to modify. The **Help** directory contains two special files with names **TableOfContents.rtf** and **Index.rtfd**. Other files in the directory are RTF (or RTFD) files containing help text specific to your application.
The **TableOfContents.rtf** (TOC) file should contain one entry for each help text file in the Help directory. It is important that this one to one mapping of TOC entries to files be maintained. The first thing on each active line should be a link to whatever file corresponds to the entry (more on links below). Following that is the text of the TOC entry which may wrap and span several lines. The TOC looks like it is displayed using a Matrix, but actually it is a modified Text object. You can

therefore use the full generality of RTF to format your table of contents.

The **`Index.rtfd`** file is structured similarly although there is no one to one mapping enforced. Generally the link that starts an index entry will specify a file and a marker within that file to go to.

The Text object for 3.0 has been modified to use hypertext-like links. Edit has been modified to allow the insertion of links and markers. A link consists of a filename and optional markername and is displayed as a little diamond embedded in the text. When the user clicks the diamond the help panel (or Edit for that matter) will display the specified file. If a destination marker is specified as part of the link it is scrolled into view and the text from the marker to the end of the line will be selected. Unless explicitly shown in Edit the actual markers are not displayed in the text. You can inspect both links and markers in Edit by holding down the Command key while clicking them.

In the template that ProjectBuilder copies into you English.lproj you will initially only have a **TableOfContents.rtf** and and an **Index.rtfd** file. If you use Edit you inspect the links in the TOC you will see that they link to files that are not found in the template. If you try to follow those links by clicking them, Edit will say that the files cannot be found. This is because the HelpPanel makes use of a search path to locate files to display. It will first look in your Help directory. If the file is found there it will be displayed. If the file is not found the HelpPanel looks in a special

compressed store file /usr/lib/NextStep/Resources/{language}.lproj/Help.store to see if the file can be located there. This pathing allows NeXT to supply standard help about getting started and for some user inferface objects. If you do not want to have this standard help in you application you may delete the unappropriate TOC entries. You may also customize copies of the standard files if you so desire. To do this use the HelpBuilder panel in InterfaceBuilder to display the file (Edit will not be able to find it since it doesn't perform the pathing). Then select all the text and copy and paste it into a new document. Save the document in your Help directory under the same name as is shown in the HelpBuilder panel. Be sure to size the window to the same width as the original so that the text will wrap the same.

The click for help feature allows a user to click on some object on the screen to get help on it. If the Help modifier key is pressed the cursor changes to a special help cursor, indicating you can then get help by clicking on some UI object (on keyboards without a Help key, you can hold Control and Alternate simultaneously to click and get help. You will generally used the HelpBuilder panel and Help inspector in InterfaceBuilder to specify which help gets displayed when a given object is help clicked.

You can also attach a help file to a user interface object programatically. You need to send the message:

```
- attachHelpFile:(const char *)filename
      markerName:(const char *)markername
```

```
to:object
```

to the **NXHelpPanel** factory. The arguments ***filename*** and ***markername*** are the same as when creating a link. The **object** argument is the object that the help is attached to. There is also a corresponding **detachHelpFrom:*object*** message to remove help from that object.

There is an exception to the above stated rule that there must be a one-to-one correspondence between TOC entries and help files in the Help directory. It is possible to have a set of files within a "hidden files" directory that do not have TOC entries. Since these files cannot be accessed from the TOC, the only way for the user to view them is by finding them with the Find command or linking to them from the Index or some other file. An example of this in the NeXT supplied help are the files in the **Objects** directory. These files are designed to be accessed by help-clicking the appropriate user interface object. It is necessary to have a TOC entry for highlighting whenever a file in a given hidden files directory is displayed. In the NeXT supplied help that TOC entry is entitled **Commands, Panels and Buttons**. The TOC entry for a hidden files directory must link to a special file **Prolog.rtfd** within that directory. This file will be displayed if the user clicks the TOC entry. The prolog file in the NeXT help says that you can get help on any command panel or button by help-clicking that object.

It is possible for your application to provide context sensitive help. The help

displayed when the **Help...** menu item is choosen will be dependent on the state of the application. To do this your application will need to initialize the panel to display the appropriate help information. This may be done by having the Application delegate respond to the **`app:willShowHelpPanel:`** method. The method should do something like the following:

```
- app:sender willShowHelpPanel:panel
{
   char path[MAXPATHLEN + 1];

   sprintf (path, "%s/%s", [panel helpDirectory],
   "Tasks/AddressingMail/CreatingAddressBook.rtfd");
   [panel showFile:path atMarker:NULL];
   return self;
}
```

In this case you must specify a fully qualified path in this case. This is because with partial paths the HelpPanel assumes the path is relative to the currently displayed help file. Since you want the code to work independently of what file is currently being displayed you should always give a full path.

The Find command makes use of the Indexing kit to quickly locate files containing the word in question. For the Find command to work quickly you should create indexes for your help files using the Indexing kit (Note: this index is different from the Index.rtfd that the user will see). If you have a hidden files directory you

should first make an index of it before creating an index of the entire Help directory. That way the hidden files will not be indexed in the main index. To create the index, in a shell, change the working directory to the directory you want indexed. Then type `ixbuild -v` with no other arguments. A .index.store file will be created in that directory for you.

- **NXPrinter**

NXPrinter provides access to static information on printers. It is a substitute for some of the **prdb** library functions. It also provides access to printer-specific information, like page imageable regions, whether a printer does manual feed, possible resolutions, etc, using PPD files. This object returns information in NetInfo and PPD files, it does not provide dynamic information about the printer state like whether the printer is running, whether it is out of paper, etc.

PPD files are now supported through the **NXPrinter** object. These are newer versions of the .pdf files found in /usr/lib/NextPrinter/pdf in older releases. The new extension is .ppd. They are now searched for in : **/NextLibrary/PrinterTypes, ~/Library/PrinterTypes, /HostLibrary/PrinterTypes, and /LocalLibrary/PrinterTypes**. The spec for these files, and most of the files we ship, are provided by Adobe. These files adhere to a newer PPD format specification than the older pdf files. They describe printer-specific information (like whether the printer is color, paper

imageable regions, etc), and printer-specific pieces of PostScript (like setting particular resolutions). For compatibility purposes, /usr/lib/NextPrinter/pdf is also searched for .pdf files.

New Methods

New methods (providing incremental functionality) added to existing classes:

- NXColorPanel

  The following methods have been added to NXColorPanel:

```
+ (void)setPickerMode:(int)mode
+ (void)setPickerMask:(int)mask
- (BOOL)doesShowAlpha
- (float)alpha
- (BOOL)isContinuous
- (int)mode
```

  `setPickerMode:` allows apps to set the ColorPanel's initial picker mode (what picker will initially be visible), without actually instantiating the ColorPanel. The **`setPickerMask:`** method is similar to the old **`setColorMask:`** method. It takes as a parameter one or more (*or*d together) of the the picker masks defined in NXColorPanel.h. This determines which pickers will be available in the

ColorPanel. This method only has an effect before the ColorPanel is instantiated. **doesShowAlpha** returns YES if the ColorPanel has an opacity slider. The **alpha** method allows one to find out the current alpha level of the Color Panel based on its opacity slider. The **isContinuous** method returns whether or not colors are currently being set continuously. The **mode** method return what picker mode the Color Panel is currently in.

- NXColorWell

  The following methods have been added to NXColorWell:

  ```
  - setBordered:(BOOL)flag
  - (BOOL)isBordered
  ```

  These enable the creation of color wells without a border (like the one in the NXColorPanel).

- TextField

  The following methods return the nextText and previousText instance variables of the receiver:

  ```
  - nextText
  - previousText
  ```

TextField now overrides the cell method, **drawCellInside:**. Additionally, TextFields now ensure that their opaque state matches that of their cells.

- drawCellInside:*aCell*

· Application

You can miniaturize all document windows in an application (i.e. all windows which appear in the Windows menu) via the following new method:

- miniaturizeAll:*sender*

Two methods have been added to suppress the normal window ordering and activation behavior that happens when the user clicks on a window, so that the source window doesn't jump to the front when the user tries to drag an item from it to another window. If a view has items that the user can drag to another window, the view should implement **shouldDelayWindowOrderingForEvent:** to return YES if the given event might initiate a drag. This method is called for every mouse down event that the view receives. Returning YES causes the normal window ordering and activation behavior that is performed in response to a mouse down to be delayed until the mouse is released. If the user does in fact drag the clicked on item, **preventWindowOrdering** should be sent to prevent the pending ordering

behavior from happening on the mouse up. Most applications will not need to send this message because the **dragImage:** method sends it when it initiates a dragging session.

```
- (BOOL)shouldDelayWindowOrderingForEvent:(NXEvent *)theEvent
- preventWindowOrdering
```

The following method should be used to indicate whether or not your application can deal with alpha components in the colors it receives from the user, or other applications (it has no effect on internal programmatic manipulations of colors). If this value is set to **YES**, then all colors your application receives (through color wells, ColorPanel, etc.) will be guaranteed to contain an alpha component, between 0 and 1.0. If you set this value to **NO**, then every color your application receives will have an alpha component of **NX_NOALPHA**. Additionally, if this flag is set to **NO**, if a ColorPanel is shown it will not have an opacity slider. The ColorPanel method **setShowAlpha:** can reverse the effects of the **setImportAlpha:** method, and the **setImportAlpha:** method can reverse the effects of **setShowAlpha:**.

The default state is **NO**, do not use alpha.

```
- setImportAlpha:(BOOL)flag
```

The following method returns the current, application-wide, "importAlpha" state, as

set by either **`setImportAlpha:`** or NXColorPanel's **`setShowAlpha:`** method.

```
- (BOOL)doesImportAlpha
```

- Workspace Protocol

  Instead of using Speaker to communicate to the Workspace Manager (to do things like open files, etc.), in 3.0 you send the messages to an object which responds to the **`NXWorkspaceRequest`** protocol found in **`workspaceRequest.h`**. These methods have the same semantics as they did before 3.0, they are just in a nicer (not restricted by Speaker/Listener argument types) form. You get this object by sending the message **`workspace`** to the Application factory.

  Example of asking the Workspace for the icon for the file "`/x.draw`":

  ```
  NXImage *i = [[Application workspace] getIconForFile:"/x.draw"];
  ```

  As part of this protocol, there are two new application delegate methods for finding out when various media (usually floppies or opticals) are mounted and unmounted. These complement the already existing **`unmounting:ok:`** method which is sent just before a device is unmounted so that applications can end all their accesses to that device. These are sent after a device has been mounted or unmounted.

```
- (int)app:sender unmounted:(const char *)fullPath
- (int)app:sender mounted:(const char *)fullPath
```

To receive these messages, you must send **`beginListeningForDeviceStatusChanges`** to [Application workspace].

Further, if you wish to hear about changes in the status of other applications, send the message **`beginListeningForApplicationStatusChanges`** to **`[Application workspace]`** and your application delegate will receive:

```
- app:sender applicationWillLaunch:(const char *)appName
- app:sender applicationDidLaunch:(const char *)appName
- app:sender applicationDidTerminate:(const char *)appName
```

Finally, you can ask the Workspace to perform a file operation for you via the **`[Application workspace]`** method ...

```
- (int)performFileOperation:(const char *)operation
          source:(const char *)source
     destination:(const char *)destination
           files:(const char *)files
         options:(const char *)options
```

This method returns 0 if the operation is synchronous and completed successfully. It returns a negative number if it fails. And if it returns a positive integer, that is a

tag which will be sent back to you when the operation completes via the delegate method:

```
- app:sender fileOperationCompleted:(int)operation
```

- Font

  These methods return a particular size of the user's default font , the user's default fixed pitch font, the System Font, and the Bold System Font.

  If fontSize is 0, the defaults database is used to determine the size. Otherwise the passed value is used. fontMatrix is used as in the other Font new methods.

  Developers should use **`userFontOfSize:matrix:`** to init their new documents to the font the user has chosen in Preferences.

```
+ userFontOfSize:(float)fontSize
    matrix:(const float *)fontMatrix;
+ userFixedPitchFontOfSize:(float)fontSize
    matrix:(const float *)fontMatrix;
+ systemFontOfSize:(float)fontSize
    matrix:(const float *)fontMatrix;
+ boldSystemFontOfSize:(float)fontSize
    matrix:(const float *)fontMatrix;
```

The font objects returned by these methods, when archived out will be replaced upon unarchiving by the system (or user or bold system or user fixed pitch) font *in effect at the time of unarchiving*.

The following methods have been added to Font to allow an app to set the user's default font & the user's default fixed pitch font for that application:

```
+ setUserFont:(Font *)aFont;
+ setUserFixedPitchFont:(Font *)aFont;
```

- FontManager

It is now possible to filter the fonts which appear in the FontPanel via the FontManager delegate method:

```
- (BOOL)fontManager:sender
    willIncludeFont:(const char *)fontName;
```

The FontManager is also now subclassable. To do this, you must call the method:

```
+ setFontManagerFactory:factoryId;
```

before loading your initial InterfaceBuilder file (which will, more than likely, try to instantiate the shared FontManager).

Two methods have been added to FontManager to round out all the possible conversions that can be made to a font via all the existing user-interface elements which can be used for that purpose (the FontPanel and the Font menu). By subclassing and overriding these (and the other convert: methods), you can track what modifications the user is making to fonts (for scripting or other purposes).

- `convert:`*`fontObj`* `toSize:(float)`*`size`*`;`
- `convert:`*`fontObj`* `toFace:(const char *)`*`typeface`*`;`

· NXBrowser (and Matrix)

You can get the selected cells in a Matrix or NXBrowser by calling:

- `(List *)getSelectedCells:(List *)`*`aList`*`;`

If *`aList`* is nil, the Matrix or NXBrowser will create a list for you.

Horizontal movement in the NXBrowser can now be done via a horizontal scroller rather than the left/right buttons on the left of the NXBrowser. There are six new methods to support his functionality:

- `scrollViaScroller:`*`sender`*`;`
- `updateScroller;`

```
- setHorizontalScrollerEnabled:(BOOL)flag;
- (BOOL)isHorizontalScrollerEnabled;
- setHorizontalScrollButtonsEnabled:(BOOL)flag;
- (BOOL)areHorizontalScrollButtonsEnabled;
```

The first is the target/action method sent by the scroller to the NXBrowser. The second can be used to redraw the scroller so that it accurately reflects the number of columns and firstVisibleColumn of the NXBrowser. The next two turn the horizontal scroller on and off. The last two are a replacement for **`hideLeftAndRightButtons:`** which is now obsolete.

The Matrix/NXBrowser methods **`allowEmptySel:`**, **`allowMultiSel:`** and **`allowBranchSel:`** have been renamed and have been complemented by methods to get the value of these various attributes. setEmptySelectionEnabled: has been added to NXBrowser.

```
- setMultipleSelectionEnabled:(BOOL)flag;
- (BOOL)isMultipleSelectionEnabled;
- setBranchSelectionEnabled:(BOOL)flag;
- (BOOL)isBranchSelectionEnabled;
- setEmptySelectionEnabled:(BOOL)flag;
- (BOOL)isEmptySelectionEnabled;
```

The method:

```
- acceptArrowKeys:(BOOL)flag;
```

Has been renamed to support sending an action when navigating through the NXBrowser using the arrow keys:

```
- acceptArrowKeys:(BOOL)flag andSendActionMessages:(BOOL)sFlag;
```

A Control standard method was missing from NXBrowser. It has been added:

```
- selectedCell;
```

- NXColorPicking

NXColorPicking is the protocol which all custom picker objects which are inserted into the ColorPanel must conform to. Some, but not all of these methods have default implementations in the NXColorPicker class. Custom color pickers occupy the area in the ColorPanel which is below the "picker mode" buttons, and above the opacity slider. The definition of this protocol may be found in the file "colorPicking.h".

Custom color pickers are installed by 1) having a directory within the application's "app wrapper" named "ColorPickers". Within this directory, there should be a separate directory for each custom picker to be installed. These directories should have the extension ".bundle", and should contain, whatever resources are

necessary for that particular picker (e.g. nib files, .tiff images, etc.). The prefix name of the directory should be the name of a class defined by the application, which conforms to the NXColorPicking protocol, and which has the same name as the prefix of the .bundle directory. This class should also be a subclass of "NXColorPicker". For instance, if an application wants to install a custom picker named "DiagramPicker", then in the .app directory of the application (Diagram.app) there would be a directory named "ColorPickers", and within the ColorPickers directory there would be a directory named "DiagramPicker.bundle". When the ColorPanel is instantiated, it will search for all the .bundle files found in the app's ColorPickers directory. For each .bundle found, it will instantiate (allocate) an object with the class name of the .bundle directory, and then send that newly allocated object an **`initFromPickerMask:withColorPanel:`** message. So, in this example, the ColorPanel will essentially do:

```
customPicker = [[MyPicker alloc]
                initFromPickerMask:someMask
                withColorPanel:self]
```

If no such class is found in the application, loading of the custom picker will be aborted. Once the controlling object has been instantiated, it will be sent a number of messages from the ColorPanel. These messages will continue to flow throughout the ColorPanel's lifetime. The custom picker can communicate to the ColorPanel using the ColorPanel's public API. If the custom picker needs to load .nib files, etc. (from its .bundle directory), or perform other initialization code,

the **initFromPickerMask:withColorPanel:** message is one time to do that. The other time do do initialization (preferred, because it's lazy), is the first time the **provideNewView:** message is received. See below for more detail.

The following methods are implemented, and derive default behavior from the NXColorPicker class:

This message is sent to installed objects when the ColorPanel is initializing. This method is the designated initializer of NXColorPicker, and of all custom color pickers. This notifies the picker of the mode mask specified by the caller of the ColorPanel method, **setPickerMask:** (or the default mask, **NX_ALLMODESMASK**, if it has never been explicitly set). The return value of this init methods is an indication of whether or not your picker supports any of the modes listed in the mode mask. If your picker supports any of the bits in the mask, return "self", otherwise, return **nil** (default NXColorPicker return value is "self", since if the **setPickerMask:** method is never called, the ColorPanel will just start off with the normal picker modes, of which your custom mode will not be a part, but by returning "self", the ColorPanel thinks you are part of the mask, and will continue to load this custom picker). This method can be used to turn off some (or all) of your subpickers, if you have any (like sliders), by seeing whether or not they are included in the mask. ***owningColorPanel*** is the id of the ColorPanel, which can be saved away for future use (this is normally done by the super's initFromPickerMask method).

This method also allows for object initialization, though most initialization (loading .nib files, etc.) should be done lazily, the first time **`provideNewView:`** is called.

This method rarely needs to be overriden, however, if it is, it should first call **`initFromPickerMask:withColorPanel:`** in super, as this is when the instance variable "colorPanel" will get set, from the passed in ***`owningColorPanel`***.

```
- initFromPickerMask:(int)mask withColorPanel:owningColorPanel
```

The following message is sent to installed objects to request the name of the image they want loaded, for subsequent installation in their mode button in the ColorPanel. An NXImage should be returned by this method. The default action of NXColorPicker is to return an image found by loading a .tiff file from the picker's .bundle which has the name "PickerClassName.tiff" (in our above example it would be called "DiagramPicker.tiff"). This method rarely needs to be overriden.

```
- provideNewButtonImage
```

The following message is sent to installed objects to have the image, ***`newButtonImage`***, inserted into the mode button for this picker. This provides a mechanism for special treatment, e.g. scaling the image first, etc. The default

action of NXColorPicker is to insert this image in the button without any modification, using **`setImage:`**. This method rarely needs to be overriden.

- `insertNewButtonImage:`*`newButtonImage`* `in:`*`newButton`*

The next message is sent to installed objects when the ColorPanel view size changes. NXColorPicker does nothing when it receives this message, other than returning "self". This method should be overriden by custom pickers if they have special preparation to do when they are resized. This method should be overridden, rather than **`superViewSizeChanged:`** type methods. ***`sender`*** is the ColorPanel.

- `viewSizeChanged:sender`

The next message is sent to installed objects when the ColorPanel has been told to add or remove the alpha (opacity) slider. Upon receiving this message, custom pickers can determine the current state of the opacity slider by asking the ColorPanel (i.e. **`[sender doesShowAlpha]`**). The default action of NXColorPicker is to simply return "self". This method rarely needs to be overriden.

- `alphaControlAddedOrRemoved:`*`sender`*

The **`insertionOrder`** method is sent to installed objects immediately after

loading. The number returned by this method indicates the position in the ColorPanel's list of "picker mode" buttons where this picker's button should be installed. The standard pickers which come with the system have the following numbers:

WheelPicker	0.50
SliderPicker	0.51
CustomPalettePicker	0.52
List Picker	0.53

These values are #defined in NXColorPanel.h. The default action of NXColorPicker is to return 0.4, meaning that the custom picker will be inserted before the other pickers. If The picker is to be inserted, say, between the wheel and the slider picker, it could return 0.501.

```
- (float)insertionOrder
```

The next message is sent to installed objects when the ColorPanel has been told, through its API (i.e. the ColorPanel method **`attachColorList:`**) to add a ColorList. This notifies the picker of this. This message is used, for example, by the provided list mode picker, so that if a ColorList is added to the ColorPanel, that new ColorList will appear in the ColorList picker. The default action of NXColorPicker is to simply return "self". This method rarely needs to be overriden.

- attachColorList:*colorList*

The next message is sent to installed objects when the ColorPanel has been told, through its API (i.e. the ColorPanel method **detachColorList:**) to remove a ColorList. This notifies the picker of this. This message is used, for example, by the provided list mode picker, so that if a ColorList is removed from the ColorPanel, the removed ColorList will be removed from the ColorList picker. The default action of NXColorPicker is to simply return "self". This method rarely needs to be overriden.

- detachColorList:*colorList*

This message is sent to installed objects when the ColorPanel has been told, through its API (i.e. the ColorPanel method **setMode:**) to change the picker mode it is in. This notifies the picker of this. This message is used, for example, by the provided slider mode picker, which has submodes, so that it knows what submode to switch to. The default action of NXColorPicker is to simply return "self". This method rarely needs to be overriden, as most ColorPickers only have one mode.

- setMode:(int)*mode*

The following methods MUST be implemented by the custom picker:

If your custom picker supports the mode specified by ***mode***, return **YES**, otherwise return **NO**. This method is called as a result of the ColorPanel being told, through its **setMode:** API, to switch modes. It is also called when the ColorPanel first starts up, and tries to restore the user's last used mode.

```
- (BOOL)supportsMode:(int)mode
```

Your custom picker should return an integer value that uniquely identifies what mode it is. This may be any integer value of your choice, though it normally should not conflict with one of the predefined and supplied modes, listed in NXColorPanel.h. This is the same integer which may get passed to you on a **supportsMode:** or **setMode:** message.

```
- (int)currentMode
```

The following message is sent by the ColorPanel when it is ready to display this picker. This might happen when the user switches pickers, when the ColorPanel first comes up, or when the mode is switched through the API. The ColorPanel also passes in a flag indicating whether or not this is the first time this object has been sent this message. This allows objects to load their UI elements in a lazy fashion (e.g. don't load .nib files, etc., until the first time this method is called). This method should return an id of a View or View subclass which will be inserted

as a subview in the picker area of the ColorPanel. This view should typically be set to autoresize both its width and height.

```
- provideNewView:(BOOL)initialRequest
```

Finally this last message is sent by the ColorPanel when the color in its ColorWell changes (perhaps a color was dragged into the Well). Some pickers update their appearance to reflect the current color in the ColorPanel (for instance the position of slider knobs in the Slider mode of the ColorPanel), so this method provides them with a notification that it is time to update their current color. This method will be called whenever the ColorPanel's **`setColor:`** method is called, *even if ColorPanel's setColor method is called from this ColorPicker*, so in a typical implementation of this method, a check is should is done to see if the passed in color is actually different than the color that the picker is currently displaying.

```
- setColor:(NXColor)newColor
```

- Open/SavePanel

A new method has been added to the OpenPanel which causes file packages to be treated like directories (i.e. users can go into file packages, to, for example, look for .eps or .tiff files).

```
- setTreatsFilePackagesAsDirectories:(BOOL)flag;
```

```
- (BOOL)doesTreatFilePackagesAsDirectories;
```

You can use the OpenPanel to get the name of a directory from the user by calling the following method before running the panel. If the user is choosing directories, the **`filterTypes`** of the OpenPanel are ignored and only directories will appear in the OpenPanel.

```
- chooseDirectories:(BOOL)flag;
```

You can control the ordering of files in the Open/SavePanel via the delegate method:

```
- (int)panel:sender
  compareFilenames:(const char *)file1
                  :(const char *)file2
         checkCase:(BOOL)caseSensitive;
```

Do not do this lightly since it may confuse the user to have the files in one Open/SavePanel in a different order than those in other Open/SavePanels or in the WSM (the Open/SavePanels sort files in the same order as the WSM by default). Note that this will also slow down the operation of the panel somewhat.

You can now set the specific subclass of OpenPanel or SavePanel used when people call **`[OpenPanel new]`** or **`[SavePanel new]`** in your application.

```
+ setOpenPanelFactory:factoryId
+ setSavePanelFactory:factoryId;
```

- Slider and SliderCell

  The following method sets the background of a slider to the NXImage specified by the argument. If the provided image is scalable, then the cell will resize it to fit its bounds. Otherwise the image might either get clipped or not fill the whole slider.

```
- setImage:(NXImage *)backgroundImage;
```

  The following methods allow changing the width (for horizontal sliders) or height (for vertical sliders) of the slider knob to the thickness specified by the argument. These methods are available to both Slider and SliderCell classes.

```
- setKnobThickness:(NXCoord)newKnobThickness;
- (NXCoord)knobThickness;
```

  The following methods have been added to allow developers to set Slider title text, text color, and text font. These methods are available to both Slider and SliderCell classes.

```
- setTitleGray:(float)grayVal;
- (float)titleGray;
```

```
- setTitleColor:(NXColor)newColor;
- (NXColor)titleColor;
- setTitleFont:fontObj;
- titleFont;
- setTitle:(const char *)aString;
- setTitleNoCopy:(const char *)aString;
- (const char *)title;
```

The following methods have been added to allow developers to retrieve and/or modify the text cell used by a SliderCell (and hence by a Slider) to actually draw the text, and modify the text attributes. By default, this cell is a TextFieldCell. Additional text attributes may be set by getting the text cell, and modifying it directly, then redisplaying the slider. The first method, setTitleCell, will not free any existing titleCell, it will simply return the old one. These methods are available to both Slider and SliderCell classes.

```
- setTitleCell:aCell;
- titleCell;
```

The following methods have been added to SliderCell to set/get the amount the slider moves by when ALT is held down, these methods are only p[rovided in the SliderCell class, and not the Slider class:

```
- setAltIncrementValue:(double)incValue
- (double)altIncrementValue
```

- Window

  A new delegate method has been added that is called when the user begins to drag a window.

  ```
  - windowWillMove:sender;
  ```

  New methods for setting the title and image in a miniwindow have been added:

  ```
  - setMiniwindowImage:image;
  - setMiniwindowTitle:(const char *)title;
  - (NXImage *)miniwindowImage;
  - (const char *)miniwindowTitle;
  ```

  The following method has been added to return whether flushing of the window's backing store is currently disabled.

  ```
  - (BOOL)isFlushWindowDisabled
  ```

  A method has been added that will order a window to the front of all windows in its tier, regardless of what app is active. Normally the system avoids putting windows on top of the key window, unless the window being ordered is in the same app as the key window. These checks are very important in multi-tasking system. This

method bypasses these checks, and should be used **only** in rare cases where the app knows it should put its window above all others. One case might be when two apps are working together via a private protocol, and the active app is using the other app to display some data, yet wants to remain active.

```
- orderFrontRegardless;
```

A method has been added to return the counterpart of a window. For most windows, this is the mini-window. If the mini-window has not been created, it returns nil. The counterpart of a mini-window is the corresponding normal window.

```
- counterpart;
```

Methods have been added to set and return the backing store type of a window. These should be used instead of PSsetwindowtype().

```
- setBackingType:(int)bufferingType;
- (int)backingType;
```

Methods have been added to control whether a window's app is activated when the user clicks on a window. This can be useful for windows that are primarily controlled via the mouse, that interact with other applications. For example, a Panel owned by a Service provider may contain only mouse controls. It may be

more convenient for the user to have the Service provider not activate when he uses that window, since he is repeatedly invoking the service from the already active Services client. Since this feature makes the window an exception to some of the normal rules windows follow, it should only be used in cases where it is well justified.

```
- setAvoidsActivation:(BOOL)flag;
- (BOOL)avoidsActivation;
```

The following methods allow setting minimum and maximum sizes:

```
- setMinSize:(const NXSize *)frameSize;
- setMaxSize:(const NXSize *)frameSize;
- getMinSize:(NXSize *)frameSize;
- getMaxSize:(NXSize *)frameSize;
```

These methods set/get the minimum and maximum window *frame* rect sizes. These size limits will be used only in those cases where the delegate would have been consulted with the **`windowWillResize:to:`** method; first these size limits will be applied, then the delegate will be informed and given a chance to fix the size up further. These limits are *not* consulted if the window size is changed programmatically (for instance, through the **`sizeWindow::`** or **`placeWindow:`** methods).

Interface Builder provides support for setting the minimum size of a window from the Window inspector. Due to limitations in autosizing, contents of windows and panels which contain kit objects with autosizing parameters may get screwed up when they are sized very small. Giving such windows reasonable minimum sizes will solve this problem.

The following new methods provide easy ways to save/restore window positions:

- `(void)saveFrameUsingName:(const char *)`*`name`*
- `(BOOL)setFrameUsingName:(const char *)`*`name`*

These methods let you save/restore the window's location and size. Typically you might use **`saveFrameUsingName:`** when a window is going away, and call **`setFrameUsingName:`** right after it's recreated to put it back in the same location. The name argument provides the key with which the stored information is accessed; this name is specific to an application.

`setFrameUsingName:` will redisplay the window if it is on screen and the size is changed. The restored size is validated by comparing it against the window's min and max sizes and calling the delegate's **`windowWillResize:to:`** method. The return value for **`setFrameUsingName:`** indicates if a frame was found using the provided name.

If the screen on which the window's size was saved is different than the screen on which it's restored, the window will auto-position itself to look good.

The following method makes the window automatically save its position everytime its frame changes. When called with a non-NULL name, this method first retrieves its frame using the name, and then saves it on every change. When called with a NULL name, this method stops the automatic save. This works fine for panels and other non-document windows and it is recommended that apps use this functionality to remember panel positions:

```
- (BOOL)setFrameAutosaveName:(const char *)name
- (const char *)frameAutosaveName;
```

Finally a method is provided to remove any frame information associated with a name:

```
+ (void)removeFrameUsingName:(const char *)name
```

The following two methods allow saving frame information in a string which can then be saved by the app in any suitable place (and not the defaults database, like the above methods do). The above methods work fine for panels and such but not for document windows (as it might be difficult to come up with unique names for documents). The following two allow saving position information with documents (if possible):

```
- (void)saveFrameToString:(char *)string;
- (void)setFrameFromString:(const char *)string;
```

These methods parallel the **saveFrameUsingName:/setFrameUsingName:** pair. The string passed into **saveFrameToString:** should contain at least **NX_MAXFRAMESTRINGLENGTH** characters; it will be terminated by a zero byte upon return. The app should save away the section upto the zero byte, not all **NX_MAXFRAMESTRINGLENGTH** bytes.

A method has been added to find out what bucky-bits were down when a window resize was initiated by the user. This can be used to provide advanced, somewhat hidden functionality to power users. This method should be used with care because there is no way for the user to know in advance what the bucky bits might do when resizing.

```
+ (int)resizeFlags;
```

- Text

The following changes were made to enhance ruler in the Text object for international functionality ± i.e., so it can be set to display centimeters and other units in addition to inches. the ruler looks at the **NXMeasurementUnits** default

for its initial state.

```
- (NXMeasurementUnit)setRulerUnits:(NXMeasurementUnit)unit;
- (NXMeasurementUnit)rulerUnits;
- toggleRulerUnits:sender;
```

The PageLayout panel has been enhanced to read the value of **`NXMeasurementUnit`** from the defaults database, and to use it for initialization. The PageLayout panel does not set the value, however. (Preferences will be enhanced to change NXMeasurementUnit to appropriate international defaults.)

The following methods have been added to the Text object to query the color or gray of the selection or of a run. The color or gray of a selection is defined to be the color or gray of the first character in the selection or the color the next character typed will be if the selection is a blinking cursor. Note that **`selGray`** was unimplemented in release 2.1--it is now functional.

```
- (NXColor)selColor;
- (float)selGray;

- (float)runGray:(NXRun *)run;
- (NXColor)runColor:(NXRun *)run;
```

The ability to find text has been added to the Text object:

```
- (BOOL)findText:(const char *)textPattern
      ignoreCase:(BOOL)ignoreCase
       backwards:(BOOL)backwards
            wrap:(BOOL)wrap;
```

You can also locate a help marker (see NXHelpPanel description). The method returns whether or not the marker was found and, if it was found, it selects the text associated with it.

```
- (BOOL)findMarker:(const char *)markername;
```

- NXImage/NXImageRep

  NXImage now provides a more generalized image file handling framework. Classes which follow the NXImageRep protocol and which can load images from files or streams can register themselves with NXImage, allowing NXImage to automatically create and use instances of these classes much in the same way it did with NXBitmapImageRep and NXEPSImageRep in 2.0.

  The following two methods allow registering/unregistering image reps. Requests to register a class twice or unregister a class which isn't registered will be ignored without an error:

```
+ (void)registerImageRep:(Class)imageRepClass;
+ (void)unregisterImageRep:(Class)imageRepClass;
```

A good place for classes to register themselves is the **+load** method.

The following methods will return the appropriate registered class. Note that for **imageRepForStream:** to work, the stream should be seekable; otherwise an error will be raised.

```
+ (Class)imageRepForFileType:(const char *)type;
+ (Class)imageRepForPasteboardType:(NXAtom)type;
+ (Class)imageRepForStream:(NXStream *)stream;
```

The following method returns a **NULL**-terminated list of file types which all the image reps registered with NXImage can load from. This list belongs to the NXImage and should not be freed or changed:

```
+ (const char *const *)imageUnfilteredFileTypes;
```

There is also a parallel for returning the supported pasteboard types:

```
+ (const NXAtom *)imageUnfilteredPasteboardTypes;
```

These lists contain all types directly supported by the registered image reps.

NXImage builds these lists by calling methods of the same name in all of its registered image rep classes. Thus, in order to provide NXImage with the information on what types are supported, every subclass of NXImageRep that gets registered with NXImage needs to implement the two methods **imageUnfilteredFileTypes** and **imageUnfilteredPasteboardTypes.** As is the case with NXImage, these return **NULL**-terminated lists of strings.

For instance, NXBitmapImageRep might define **imageUnfilteredFileTypes** as:

```
+ (const char *const *)imageUnfilteredFileTypes
{
   static const char *const types[] = {"tiff", "tif", NULL};
   return types;
}
```

The list **{NULL}** should be returned in case a class does not recognize any file or pasteboard types. If a subclass of a subclass of NXImageRep is willing to support the types supported by the superclass, then it should declare them in addition to its own types. This can be done by getting the types supported from the superclass and actually building and caching a local list at runtime, augmenting it with it the additional types.

There are also parallel sets of methods in NXImage and NXImageRep which return

all types that can be opened, including those accessible through the use of filter services. (See elsewhere for discussion of filter services.) These methods are:

```
+ (const char *const *)imageFileTypes;

+ (const NXAtom *)imagePasteboardTypes;
```

These methods are implemented in both NXImage and NXImageRep and there is no need for NXImageRep subclasses to implement them. Because these methods return **NULL**-terminated lists of strings, the return values can be passed directly into the OpenPanel **runModalForTypes:** method. In fact, most applications which accept images (either through the OpenPanel or through dragging) and use NXImage to open them should switch over from using a hardwired list such as **{"tiff, "eps", NULL}** to using one of these two methods.

The following NXImageRep method allows NXImage to determine if a registered image rep can load an image from a given stream. This method should look at the stream to see if it contains a valid image. The stream pointer should be left at where it was on entry:

```
+ (BOOL)canLoadFromStream:(NXStream *)stream
```

Two other new methods in NXImage and NXImageRep are:

```
+ (BOOL)canInitFromPasteboard:(Pasteboard *)pasteboard;
- initFromPasteboard:(Pasteboard *)pasteboard;
```

These methods can be used to initialize NXImages and NXImageReps from a pasteboard. Data can come from a supported pasteboard type (such as TIFF or EPS) or a file name type containing a file name which can be filtered to a supported pasteboard type.

This next method allows NXImage to load representations from a file:

```
- (BOOL)loadFromFile:(const char *)fileName;
```

This method is essentially a short cut to opening a stream on the specified file and calling **loadFromStream:**. Thus the data is loaded immediately (rather than lazily, which is what **useFromFile:** and **initFromFile:** do) and the file name is not remembered.

Note that NXImage can use filters only when typed data is availableÐfor instance when an image is loaded from a file (using methods such as **initFromFile:**, **loadFromFile:**, etc.) or from a pasteboard (with **initFromPasteboard:**). Images loaded from streams cannot be filtered as no type information is available to aid in choosing the right filter. If you load images into NXImages from streams which you create yourself, you can just switch over to using **loadFromFile:**, which will map the

file in (filtering if necessary) into a stream and use **`loadFromStream:`** to read the data in.

On a somewhat unrelated note, NXImage and NXCachedImageRep now have **`copyFromZone:`** methods. This allows NXImages to be copied.

And, finally, another new method in NXImage is the following, which allows you to specify the compression type & factor when writing an NXImage to a TIFF file. **`writeTIFF:allRepresentations:`** now calls this method.

```
- writeTIFF:(NXStream *)stream
   allRepresentations:(BOOL)flag
     usingCompression:(int)compression
            andFactor:(float)aFloat;
```

· NXBitmapImageRep

NXBitmapImageRep factory now has methods to indicate whether the app can deal with unpacked image data:

```
+ (void)setUnpackedImageDataAcceptable:(BOOL)flag;
+ (BOOL)isUnpackedImageDataAcceptable;
```

Unpacked images have non-default bytesPerRow and/or bitsPerPixel values. The

3.0 WindowServer can generate such images and NXBitmapImageRep can store them; however, such an image will never be given back to a client which is not expecting it.

An app linked against a pre-3.0 shlib is never given unpacked data; an app linked with a 3.0 shlib is assumed to be capable of dealing with it. The **`setUnpackedImageDataAcceptable:`** method or the **`NXUnpackedImageDataAcceptable`** default can be used to override this behaviour.

It's often more efficient to leave unpacked images unpacked; thus applications should pay attention to **`bytesPerRow`** and **`bitsPerPixel`** when traversing image data. When an unpacked image is written out to a TIFF file or printed a packed copy will be used (as TIFF and PostScript print jobs are not capable of dealing with unpacked data); this packing takes place automatically.

There are some constraints on the format of unpacked data; please refer to the discussion on **`NXDrawBitmap()`** elsewhere in this document and the window server notes for more information if you wish to create and manipulate your own unpacked images.

In 3.0 NXBitmapImageReps loaded from TIFF files remember their compression type. Methods are provided to change the compression:

```
- (void)getCompression:(int *)compression
            andFactor:(float *)factor;

- (void)setCompression:(int)compression
            andFactor:(float)factor;
```

writeTIFF: (without the explicit **compression:andFactor:** arguments) will use the stored compression when saving. This behaviour is changed from 2.0 where **writeTIFF:** would use no compression. The benefit is that if a compressed image is read in and later saved, its compression will be preserved. If the image cannot be written out with the compression that is provided (this could happen with compressions no longer supported, such as the NEXTSTEP 1.0 2-bit encoding), then **writeTIFF:** will use no compression.

The following methods were added to give developers information about what TIFF compression types are available and which ones are applicable to a given image. Because not all compression types can be used with all images and because future releases of NEXTSTEP may bring along other compression types, these methods should be used whenever an application wants to put up some UI to let the user choose a compression type for an image.

```
+ (void)getTIFFCompressionTypes:(const int **)compList
                          count:(int *)numTypes;
```

```
+ (const char *)localizedNameForTIFFCompressionType:(int)comp;

- (BOOL)canBeCompressedUsing:(int)comp;
```

The first method above returns a pointer to an array of ints containing all available compression types that can be used when writing a TIFF image. These compression types are currently defined in tiff.h. The list argument belongs to NXBitmapImageRep and should not be freed or altered. It points to an array of **numTypes** ints.

The **localizedNameForTIFFCompressionType:** method returns the localized name for the specified compression type. **NULL** will be returned if the compression type is not recognized. The returned string should not be freed or altered in any way.

The last method, **canBeCompressedUsing:**, allows asking an instance of NXBitmapImageRep if it can be compressed with specified compression type.

- Pasteboard

The owner of a pasteboard can implement the following method to find out when he loses ownership of the pasteboard. The owner is not able to read the contents of the pasteboard he owned when responding to this method. The owner should be

prepared to receive this method at any time, even in from within the **`declareTypes:num:owner:`** he uses to declare his ownership.

```
- pasteboardChangedOwner:sender
```

The following method has been added to deallocate data returned by **`readType:data:length:`**. This method should **always** be used to free Pasteboard data returned by that method (i.e. do **not** use **`vm_deallocate()`**):

```
- deallocatePasteboardData:(char *)data
                    length:(int)numBytes;
```

A convenience method has been added to scan the available types for one you want. It returns the first type in the ***`types`*** that you supply that is available in the Pasteboard:

```
- (const char *)findAvailableTypeFrom:(const char *const *)types
                                  num:(int)numTypes;
```

The following method allows you to add types to the list previously declared by using the existing Pasteboard method **`declareTypes:num:owner:`**. It can be useful when subclassing **`copy`** methods, or when multiple modules need to contribute data to a single **`copy`**. It should only be sent when you know that a

declareTypes:num:owner: has already been sent for the particular copy operation.

```
- (int)addTypes:(const char *const *)newTypes
            num:(int)numTypes
          owner:newOwner;
```

Another convenience method for getting data from a stream into the Pasteboard has been added. It takes the data from the stream and inserts it into the receiver under the given ***dataType***. The **stream** must be readable. If it is seekable, the **stream** is seeked back to the start before the data is read; otherwise, data is read from the current position until the end of the **stream**.

```
- writeType:(const char *)dataType
  fromStream:(NXStream *)stream;
```

A similar method get data out of the Pasteboard into a stream. *Be sure to deallocate the stream returned with* `NXCloseMemory(stream, NX_FREEBUFFER)`. You do not need to send the `deallocatePasteboardData:` message with the stream data.

```
- (NXStream *)readTypeToStream:(const char *)dataType;
```

A new method has been added to the Pasteboard which returns an instance of a

Pasteboard with a name that is guaranteed to be unique with respect to other Pasteboards on the system. This would only be used by applications doing their own IPC using pasteboards to pass data.

```
+ newUnique;
```

- View

A new method in View has been added as a convenience for copying EPS to the pasteboard. It writes the EPS representing an image of the receiver to the pasteboard under the EPS pasteboard type. If passed **NULL** for the rectangle, EPS for the entire view is generated. The caller should first use the **declareTypes:num:owner:** Pasteboard method in the standard way.

```
- writePSCodeInside:(const NXRect *)copyArea
                 to:pasteboard;
```

A method to return the autosizing parameters of a View has been added:

```
- (unsigned int)autosizing;
```

A method has been added which allows the developer to send a fax without putting up the fax panel UI:

```
- faxPSCode:sender
      toList:(const char *const *)names
  numberList:(const char *const *)numbers
      sendAt:(time_t)when
  wantsCover:(BOOL)cFlag
 wantsNotify:(BOOL)nFlag
  wantsHires:(BOOL)hFlag
     faxName:(const char *)aString;
```

- PrintInfo/PrintPanel

The **NXPrinter** object associated with the **PrintInfo** should be used to get/set information which used to be accessed via methods like **setManualFeed:**. These new methods have been added to access the **NXPrinter**:

```
- setPrinter:(NXPrinter *)pr;
- (NXPrinter *)printer;
- initializeJobDefaults;
```

PrintInfo has a number of other new features. It provides and sets the default **NXPrinter** object. New methods:

```
+ (NXPrinter *)getDefaultPrinter;
+ setDefaultPrinter:(NXPrinter *)pr;
```

Methods have been added to **PrintInfo** to specify whether the generated page order should be "normal" or "reversed". Normal page order depends on the device that is being output to. "Save" is done first-to-last, as are some printers. The NeXT Laser Printer is done last-to-first.

```
- setReversePageOrder:(BOOL)flag;
- (BOOL)reversePageOrder;
```

PrintInfo also supports printer-specific PostScript operations using PPD info. These are called "job features", and include manual feed, setting resolution, page size, and others.

```
- setJobFeature:(const char *)feature
        toValue:(const char *)string;
- (const char *)valueForJobFeature:(const char *)feature;
- setJobFeature:(const char *)feature
    toValueList:(const char *const *)list;
- (const char **)valueListForJobFeature:(const char *)feature;
- removeJobFeature:(const char *)feature;
- (const char *const *)jobFeatures;
- setPaperFeed:(const char *)str;
- (const char *)paperFeed;
```

The following two methods have been added to the PrintPanel. They can be called to update the PrintPanel from its PrintInfo (or write out the changes in the

PrintPanel to its PrintInfo).

```
- updateFromPrintInfo;
- finalWritePrintInfo;
```

- Matrix

  It is now possible to do "drag selection" in Matrix programatically via the method:

```
- setSelectionFrom:(int)startPos
               to:(int)endPos
           anchor:(int)anchorPos
              lit:(BOOL)lit;
```

  Drag selection in Matrix now works in a visual fashion (i.e. the user will drag out rectangular regions) rather than on a row-oriented basis. The old behaviour is still available, but applications must call **`setSelectionByRect:NO`** to turn off this behaviour:

```
- setSelectionByRect:(BOOL)flag;
- (BOOL)isSelectionByRect;
```

New Functions

- Alert

  A new, localizable version of `NXRunAlertPanel()` has been added to the kit:

```
int NXRunLocalizedAlertPanel(
   const char *table,
   const char *title,
   const char *s,
   const char *first,
   const char *second,
   const char *third, ...,
   const char *comment
);
```

  The **`genstrings`** program will generate an appropriate entry in the strings file *`table`* with the comment *`comment`* and translation entries for each of the items. This only works, of course, for literal string arguments and will obviously not translate the "*`...`*" arguments.

- Imaging functions

  The following function is a cover for the new window-to-window imaging feature in the WindowServer:

```
extern void NXCopyBitmapFromGState(int srcGState,
   const NXRect *srcRect,
   const NXRect *destRect);
```

Bits from the rectangle specified by ***srcRect*** in the source graphics state ***srcGState*** will be imaged into the destination rectangle specified by ***destRect*** in the current graphics state.

NXCachedImageRep's **draw** and NXImage's **setSize:** methods now also take advantage of this functionality whenever possible and thus are considerably faster under the right circumstances.

The following function, **NXDrawBitmap()**, replaces **NXImageBitmap()**. Two additional arguments (***bytesPerRow*** and ***bitsPerPixel***) are provided and the existing arguments have been cleaned up to take new-style image parameters (***colorSpace***, ***isPlanar***, and ***hasAlpha*** instead of ***photoInt*** and ***planarConfig***).

```
extern void NXDrawBitmap(
   const NXRect *rect,
   int pixelsWide,
   int pixelsHigh,
   int bitsPerSample,
```

```
    int samplesPerPixel,
    int bitsPerPixel,
    int bytesPerRow,
    BOOL isPlanar,
    BOOL hasAlpha,
    NXColorSpace colorSpace,
    const unsigned char *const data[5]
);
```

When converting any existing calls to **`NXImageBitmap()`** to **`NXDrawBitmap()`**, you can use the following mapping from the old style arguments to the new style arguments:

```
isPlanar     = (planarConfig == NX_PLANAR) ? YES : NO;

hasAlpha     = ((photoInt & NX_ALPHAMASK) != 0) ? YES : NO;

colorSpace   = ((photoInt & 3) == 3) ? NX_CMYKColorSpace :
                (photoInt & 3);

data         = {data1, data2, data3, data4, data5};

bitsPerPixel = bitsPerSample *
   ((planarConfig == NX_PLANAR) ? 1 : samplesPerPixel);

bytesPerRow  = (7 + pixelsWidth * bitsPerSample *
   ((planarConfig == NX_PLANAR) ? 1 : samplesPerPixel)) / 8;
```

The ***colorSpace*** argument to **NXDrawBitmap()** can be **NX_CustomColorSpace**, indicating that the image data is to be interpreted according to the current color space in the PostScript graphics state. This allows for imaging using custom color spaces. The image parameters supplied as the other arguments should match what the color space is expecting.

If the image data is planar, ***data*****[0]** through ***data*****[spp-1]** point to the planes; if the data is meshed, only ***data*****[0]** needs to be set.

Under 3.0, there are some restrictions on ***bytesPerRow*** and ***bitsPerPixel*** arguments; please refer to the window server notes for details. Also, any unused space within a pixel (which happens if ***bitsPerPixel*** is greater than ***samplesPerPixel*** * ***bitsPerSample***) should be filled with ones; this will be the case by default for images read back from the window server, but not images you create yourself. This requirement doesn't apply to unused space at the end of scanlines (which happens when ***bytesPerRow*** is different than the default value).

- NXColor

The following functions have been added to get the list and color names of named colors. (Please refer to the discussion on NXColorList for more information on named colors.) These functions return **NULL** if the color is not named. Under 3.0,

only lists such as PANTONE (and similar lists which maybe be created manually), generate named colors; other colors will not remember their names.

```
const char *NXColorListName (NXColor color);
const char *NXColorName (NXColor color);
```

The following function allows getting a color given a list and color name. NO will be returned if ***listName*** and ***colorName*** do not refer to any color.

```
BOOL NXFindColorNamed (
   const char *listName,
   const char *colorName,
   NXColor *color
);
```

- Reading a pixel

NXReadPixel() will read the color from the specified pixel of the currently lockfocus'ed view, for example:

```
if ([myNXImage lockFocus]) {
   NXSize imageSize;
   NXPoint centerPoint;
   NXColor centerColor;
```

```
    [myNXImage getSize:&imageSize];
    centerPoint.x = floor(imageSize.width / 2);
    centerPoint.y = floor(imageSize.height / 2);
    centerColor = NXReadPixel(&centerPoint);

    [myNXImage unlockFocus];
}
```

This function will always convert the point into screen coordinates, round down to the nearest pixel (if necessary), and then take the pixel encompassed by a 1 X 1 rectangle with the specified point being the lower-left origin.

- Services

To invoke a Services Menu services programatically, use the function:

```
BOOL NXPerformService(const char *itemName, Pasteboard *pboard);
```

It returns whether the service was successfully performed. *itemName* is a Services menu item (in any language). Note that Services menu entries which are in subdirectories must include a slash wherever there is a subdirectory, e.g., "Mail/Selection". The *pboard* must include the requisite data which feeds the services and, upon return of the function, will contain the resultant data provided by the service provider.

- DPSClient/pswrap

  Single operator wraps have been added for all level 2 operators.

  `DPSAsynchronousWaitContext()` has been added to allow a client to learn asynchronously when all PostScript code it has generated has been executed.

  Wraps that take **`numstring`** arguments will now produce PostScript that will work on all PostScript printers, so it is no longer necessary to have separate versions of these wraps for printing. **`DPSWriteNumString()`** has been added to to support pswrap's sending of encoded number strings.

  `DPSSendPort()` has been added to send a tagged port to the Window Server using **`DPSSendTaggedMsg()`**. It is used to communicate a port to PostScript operators that receive a port via that tagged message mechanism.

  `DPSAddNotifyPortProc()` and **`DPSRemoveNotifyPortProc()`** have been added to allow an application to filter messages received on the task's notify port. Procedures registered in this way should check all notify messages they are passed to be sure they pertain to the ports they are interested in. Clients that previously passed the notify port to **`DPSAddPort`** should switch to this API.

The function **DPSCreateNonsecureContext()** has been added to allow the creation of DPS contexts that are able to to write files.

A new function **DPSSynchronizeContext()**. If called with **YES**, a **DPSWaitContext()** will be performed after every pswrap that is executed. The default **NXSyncPS** is automatically passed to this function. This is useful in conjunction with **NXShowPS** when debugging PS errors, since it allows you to isolate the piece of PS causing the error.

The function **DPSSendTaggedMsg()** has been added. It is used to send a Mach message to the Window Server on behalf of a DPSContext. It is used in conjunction with certain PostScript operators that operate on Mach ports or out-of-line data. Usually a higher level interface will handle calling this function.

Semantic Changes

The following changes affect the semantics of existing functionality in the Application Kit:

- NXColorPanel

  The ColorPanel's opacity default has changed from its 2.1 default. In 2.1 the ColorPanel had an opacity slider by default. Now it does not have an opacity

slider by default, and the Application is set to "not import alpha" by default (see Application's **setImportAlpha:** method). This change has no effect on applications which have not been recompiled for 3.0.

The ColorPanel's "continuous" default has changed from its 2.1 default. In 2.1 the ColorPanel was not continuous by default. Now it is. This change has no effect on applications which have not been recompiled for 3.0.

If color panel's initial mode mask is "0" (no pickers), the panel will not be shown. This is an effective way of preventing the panel from ever being shown (even when wells in your app are clicked on, etc.).

- NXColorWell

  NXColorWell's **drawWellInside:** used to **lockFocus** and **unlockFocus**. In apps linked against 3.0, it will no longer do this.

- Panel

  The autosizing bits contained in the view passed to Panel's **setAccessoryView:** method used to be ignored, and replaced. This has changed so that if there are autosizing bits with this view, they will not be modified. If there are no autosizing bits, the **(NX_MINXMARGINSIZABLE | NX_MAXXMARGINSIZABLE)** bits will be

added. This change has no effect on applications which have not been recompiled for 3.0.

- NXImage

  Under 2.0, if an NXImage had a delegate implementing the **`imageDidNotDraw:inRect:`** error handling method, this delegate would be consulted whenever the image was told to **`composite:...`** (or **`dissolve:...`**) and the operation failed (most probably because the cache could not be created, the image did not exist, or there were errors during rendering). Under 3.0, for apps linked against 3.0, this delegate method will also be consulted when **`lockFocus`** or **`lockFocusOn:`** encounters an error. If the delegate method does return an alternate image, then that image will be sent a **`lockFocus`** and the result of that operation will be returned to the client. If either **`lockFocus`** (on the original image or on the error handler) succeeds, the client should call **`unlockFocus`** (on the original image).

  Under 2.x, all NXImages caches created as a result of **`useCacheWithDepth:`** (or **`lockFocus`** without any other image source) were unique, even through the unique bit is off by default. What's worse, even if you called **`setUnique:NO`**, they'd still remain unique. Under 3.0, by default, these caches will still be unique; however, if you do set the uniqueness one way or the other, the unique bit will be honored. Thus it's possible for simple cached images to share caches between each

otherÐJust make sure you don't accidentally promote caches (thus all other images in the caches) to have more colors or alpha while drawing into an image. Alpha promotion is the greater problem; because windows depths are normally limited, undesired color promotion won't happen too often. For instance, caches which are meant to serve as temporary backing stores for underbits of covered areas in windows should probably still be kept uniqueÐOtherwise if they are accidently promoted, they will end up promoting the original source window when composited!

This is not an issue for images created from files or streams as their depths are determined by the NXImage and appropriate caches chosen. For custom images (added to NXImage with **`useDrawMethod:inObject:`**) you should set the image characteristics of the image rep to let NXImage choose the appropriate depth cache.

Under 2.x, images upto 128 pixels wide could be put into shared image caches; under 3.0, this limit is 256 and can be set with **`NXMaxSharedImageWidth`** default. The purpose of this default is to allow developers to easily catch bugs resulting from images inadvertently sharing caches; if you think an application is having problems due to shared image caching, try:

```
dwrite appname NXMaxSharedImageWidth -1
```

Value of -1 disables all image cache sharing. During normal use an app should not

have to set this default; if an image cannot be put in a shared cache for some reason, use **`setUnique:YES`**.

The **`setSize:`** method of NXImage no longer forces a cache redraw if the new size is the same as the old size. This change will only take effect for applications that are linked under 3.0.

- Designated initializer for Window and its subclasses

  This is not a semantic change, just a reiteration that the designated initializer (the one you should override to put subclass-specific behaviour in) for Window is **`initContent:style:backing:buttonMask:defer:`**, not the same method with **`screen:`** at the end!

- Speaker/Listener

  Speaker/Listener messages are now secure (meaning you cannot get the port of a Listener on another machine unless that machine is running with PublicWindowServer turned on).

  The **`performRemoteMethod:paramList:`** method of Listener no longer requires that the method contained in the NXRemoteMethod structure come from the **`remoteMethodFor:`** method. These structures can now be composed, and

the method will be searched based on a comparison of the selector in the structure.

- Matrix

  Radiomode matrices now properly allow "Empty Selection" (i.e. the absence of any selected cell), when set to allow it. In the past, if a Radiomode matrix was set to allow empty selection, it would still never allow all cells to be deselected at once. It will now allow this. This change will only effect applications which are recompiled for 3.0.

  In list mode, the **`selectedCell, selectedRow`**, and **`selectedCol`** methods used to return the row, column and cell of the cell which received the mouse up during a tracking session. To return a more useful result, if the cell moused-up on was deslected as a result (i.e. the shift key was down), Matrix will now return the previously selected cell, if it is still selected, otherwise it will find the first highlighted cell it can, if there are no highlighted cells, it will return the cell moused-up on. This fix only effects apps which recompile post-2.1.

  When the Matrix's **`selectAll:`** method is called, each cell in the Matrix will have their **`highlight:inView:lit:`** method called. Before 3.0, only their state was set to 1. This change will only take effect in applications relinked under 3.0.

  According to the documentation, **`selectCell:`*`aCell`*** was supposed to return **`nil`**

if **aCell** is not in the receiving Matrix. In actuality, it would return whatever value was contained in **aCell**. For applications relinked under 3.0, this has been changed to behave as described in the documentation (e.g. return **nil** if **aCell** is not in the receiving Matrix).

According to the documentation, **findIndexWithTag:*aTag*** was supposed to return **-1** if there is no field with **aTag** in the receiving Form. In actuality, it would return 0. For applications relinked under 3.0, this has been changed to behave as described in the documentation (e.g. return **-1** if a field with **aTag** is not in the receiving Matrix).

The **insertRowAt:** and **insertColAt:** methods have been changed so that if the column or row where the insertion is to take place is larger that the last row or column, then all necessary rows or columns will be created in order to accomodate the request. This sort of request used to cause a segmentation fault. This is now a useful method for increasing the size of matrices by more than one row or column at a time.

The semantic for sending a **doubleAction** in Matrix has been changed. In 2.1, if a Matrix received a double-click, it would first try to send it to the cell's **action** method. If the cell had no **action** method, it would use the Matrix's **doubleAction** method. If the Matrix had no **doubleAction** method, it would

use the Matrix's **action** method. Applications linked under 3.0 will have the following semantic: the Matrix will first try and call the **doubleAction** method associated with itself. If and only if there is none, then it will try calling the cell's **action** method. If that is not set, it will call its own (Matrix's) **action** method.

- Text object delegate method **textWillEnd:**

  In 2.x, the Text delegate would receive the **textWillEnd:** method immediately before the Text object resigns first responder, but only if the Text object had been changed since last becoming first responder. Apps linked against the 3.0 library will receive this action, regardless of whether the Text object has changed. To detect if the text has in fact changed, the delegate must respond to **textDidChange:** and remember this fact when **textWillEnd:** is sent to the delegate.

- NXPortFromName()

  When passed **NULL** or "" as a hostname, this function will now try to look up the port on the host specified by the **NXHost** default, if the default has been set. Otherwise it will look on the local host.

- NXImageRep

Under 2.0 and 2.1 the **drawAt:** and **drawIn:** methods could change the graphics state under certain circumstances. (Mostly by design and in one case, NXEPSImageRep's **draw**, due to a bug). In apps linked against 3.0, this is no longer the case; these methods always leave the graphics state as they found it.

In 2.x, the **write:** methods of NXBitmapImageRep and NXEPSImageRep saved either the filename or the data depending on whether the image rep was created from a file or stream. This made them behave more like NXImage. Under 3.0, for apps linked against 3.0, the **write:** method of these classes always write out the data, whether or not the original data came from a file or a stream.

Note that NXImage still behaves as it did under 2.x; the **write:** method will write enough data out to recreate the image the same way it was originally created (from a name, from a file, from a section, or from actual data). This behaviour follows the overall design of NXImageÐto be lazy and to hold on to as little redundant information as possible. If you wish to create an NXImage from a file but actually have the data archived when you **write:**, use the **loadFromFile:** method instead of **useFromFile:** or **initFromFile:**. NXImages which are handed filenames through **useFromFile:** or **initFromFile:** firmly (and perhaps naively) believe that those files will always be around.

- Control/Cell

If you send **setFloatValue:** or **setDoubleValue:** to a Cell (or Control), the AppKit will use the appropriate thousands separator depending on the language of the user (usually either "," or "."). When the user types into a Cell (whether it be a TextField or Form or whatever), it will accept EITHER the local separator or, for backwards compatibility, ".". This only occurs in applications which recompile post-2.1. Note that **floatValue** and **doubleValue** will still return valid values.

The ButtonCell method **setIconPosition:** used to inconsistently update itself. It now always updates itself (for applications relinked under 3.0).

The Button and ButtonCell method **setAltImage:** used to only remove the altImage when the "image" parameter was nil IFF there was a "regular" image. Now, if "image" is nil, the altImage will be removed (for applications relinked under 3.0).

setEditable: in Cell no longer forces a **setSelectable:** (for applications relinked under 3.0). The original selectability is unaffected by calls to **setEditable:NO**.

- NXBrowser's setCellPrototype: and Matrix's setPrototype:

  The semantics of **setPrototype:** in Matrix is that the Matrix "owns" that prototype and will free the prototype when the Matrix itself is freed. In 2.1

NXBrowser abused this semantic by giving the same prototype object out to many matrices and then protecting them when the Matrix's were freed so that there would not be multiply freed objects. This was incorrect.

In 3.0, the NXBrowser now copies (by sending **`copy`** to the cell) the prototype cell passed to it via **`setCellPrototype:`** before giving it to its Matrix's via **`setPrototype:`**. This also means that the NXBrowser no longer protects against the case where its Matrix's share a single prototype cell. This protection is left in for applications linked under 2.1, but will go away when you rebuild your application under 3.0 (otherwise the Matrix's would leak their prototype cells).

Thus, if you set the prototypes of the Matrix's in an NXBrowser DIRECTLY (i.e. not via **`setCellPrototype:`** in NXBrowser), such that more than one of the Matrix's have exactly the same prototype cell (and not copies thereof), your application will crash when you rebuild it under 3.0. The fix is simple, either use **`setCellPrototype:`** in NXBrowser, or, if you find you must set the prototypes directly in the Matrix's, use **`[matrix setPrototype:[myProto copy]]`** so that they each get their own copy and don't share a single object.

- NXBrowser

  It is legal now (if the NXBrowser's delegate implements

loadCell:atRow:inColumn:) to pass nil for the NXBrowser's **prototype** or **cellClass**. This will cause the NXBrowser to never allocate any cells in the columns. This means that every time any information is needed about a cell, the delegate will be consulted. This is obviously somewhat more time consuming that caching the information in cells in the matrix, but for extremely long lists (100,000s, 1,000,00s or more), the space/time trade-off may be worth it. Notice that an NXBrowser like this must be treated with care because there are no cells in the matrices, thus normal activity (like selecting ranges of cells, for example) is sometimes not possible.

- NXColor

  The **NXSetColor()** function now uses the new window server operator **nxsetrgbcolor** to set its color when in any modes other than CMYK. **nxsetrgbcolor** uses the **NXCalibratedRGB** color space which is closely related to colorimetry of NeXT monitors; thus it is fast and suitable for interactive rendering. (Please refer to the window server notes for details.)

  Because **nxsetrgbcolor** changes the color in a color space other than **DeviceRGB**, apps setting the color through **NXSetColor()** and then reading it back from the server with operators such as **currentrgbcolor** or its friends will fail to get correct results. A default, **NXUseCalibratedColor**, has been provided to disable the use of the **NXCalibratedRGB** color space in NXColors for such

apps:

```
dwrite appname NXUseCalibratedColor NO
```

Note that one side-effect of all these changes is that when printing the output that the kit produces relies more on the standard print package than it did in 2.0. (The "standard print package" is the printPackage.ps file which gets included with every print job or EPS file generated by the kit.) For instance, **`NXSetColor()`** now generates **`nxsetrgbcolor`** operator and assumes it has been defined appropriately in the print package. The standard print package does indeed contain a definition for this operator; the definition is conditional on whether the printing device is Level I or Level II. However, if the print package has not been included for some reason, this operator will generate PostScript errors on non-NeXT printers.

The bottom line is that *any app which relies on the kit objects in any way to generate PostScript should go through the View machinery when generating EPS files and printing*. A 2.1 application which doesn't do this might not be able to print on non-NeXT printers under 3.0; in that case, the **`NXUseCalibratedColor`** default can be used as shown above to get the app to print.

As mentioned earlier, by default, the kit will color calibrate the print output from old applications (applications linked with a version of NEXTSTEP older than 3.0) by interpreting **`setrgbcolor`**, **`setgray`**, **`sethsbcolor,`** and **`colorimage`** as

calibrated. When a 3.0 application prints, it is assumed that the application is aware of color calibration issues; thus the output (except where EPS files are imported) will not be calibrated. Thus even if an app cannot use NXColors, it should still try to generate calibrated colors. This can be accomplished by using the **NXCalibratedRGB** colorspace or the **nxsetrgbcolor** and **nxsetgray** operators, which are defined both when drawing to the display and when printing (the latter is true only when the printPackage is imported).

- PostScript Level II

  The kit does not provide Level I emulations of various PostScript Level II operations; thus code which uses patterns, custom colorspaces, etc might not print on Level I printers. However the Kit assures that its own use of Level II operators (such as the **NXCalibratedRGB** color space or patterns for scroller backgrounds) will work on Level I printers. Thus NXColor is safe in all cases and **NXDrawBitmap()** is fine as long as long as a custom color space isn't utilized (through **NX_CustomColorSpace**).

  The following snippet shows how an application can make sure that any Level II PostScript code it generates is conditional on the **languagelevel** operator so that the PostScript file can be interpreted without errors on both Level I and Level II devices:

```
/DeviceIsLevel2
   systemdict /languagelevel known
   {languagelevel 2 ge}{false}ifelse
def
```

- OpenPanel/SavePanel

  When you **`runModalForTypes:...`** in the OpenPanel, and one of the "types" you were running modal for was not owned by an application in your application paths (e.g., **`~/Apps`**, **`/LocalApps`**, etc.), and documents of that type were file packages, they would incorrectly show up as directories in the OpenPanel. This has been fixed.

  The validation method of SavePanel used to be called twice sometimes. It is only called once now (the double call has been left in for compatibility with 2.0 applications--once you recompile your application under 3.0, that behaviour will go away).

  When running Save or Open panels with "runModalForDirectory", if the passed in directory string is more than a directory, e.g. it has a filename at the end of it, that filename will be honored, and selected (this was true for 2.1 as well). If a "file:" parameter is specified as well, it will take precedence (if valid) over any trailing filename found in the passed in directory. This will only effect 3.0 apps. As an example, assume there is a directory "/tmp", and off it are two files, "console.log"

and "dummy", here's what happens under various circumstances:

[myPanel runModalForDirectory:"/tmp" file:"dummy"] <- /tmp/dummy are highlighted

[myPanel runModalForDirectory:"/tmp" file:"duGmmy"] <- /tmp is highlighted

[myPanel runModalForDirectory:"/tmp" file:""] <- /tmp is highlighted

[myPanel runModalForDirectory:"/tmp/dummy" file:""] <- /tmp/dummy are highlighted

[myPanel runModalForDirectory:"/tmp/console.log" file:""] <- /tmp/console.log are highlighted

[myPanel runModalForDirectory:"/tmp/console.log" file:"dummy"] <- /tmp/dummy are highlighted (dummy takes precedence over console.log)

[myPanel runModalForDirectory:"/tmp/console.log" file:"duGmmy"] <- /tmp/console.log are highlighted (duGmmy doesn't exist).

If users of "chooseDirectories:YES" want file packages to show up as well, then they should also call "setTreatsFilePackagesAsDirectories:YES".

- Delayed Perform change

The **`perform:afterDelay:cancelPrevious:`** method used to (under some circumstances) perform the method twice if you passed 0 for the delay. That is no longer true. Passing a negative number for the delay now means "don't perform it (just **`cancelPrevious:`** if specified)." Passing zero means perform it as soon as possible, but not now (i.e. after the application returns to getting events from the user). This behaviour will only start to happen after you relink your application under 3.0.

- NXHomeDirectory() and NXUserName()

  The **`NXHomeDirectory()`** and **`NXUserName()`** functions operated on the real uid of the process in 2.0. After relinking in 3.0, the semantics will change to operate on the effective uid (euid) of the process unless the euid of the process is zero, in which case, the real uid will be used.

- Window

  If a window has to be moved or resized when being ordered front to fit a given screen, the window delegate will be sent **`windowDidMove:`** or **`windowDidResize:`**. (In 2.1 this was not the case; the delegate would not be informed.) Along the same lines, if a window has to be resized to fit on the screen, the delegate's **`windowWillResize:to:`** method will be called one or more times to determine if the window can be resized and what an acceptable size is.

- Application sections

  All methods in the AppKit which use look in the macho section to find something (e.g. a **`.nib`** file, an EPS or TIFF file, or a **`.snd`** file), now look in **`[NXBundle mainBundle]`** if the thing being searched for is not found in the macho section. The effect of this is to get "for-free" localization of these files simply by making your application an app-wrapper and putting all of your **`.nib`**, **`.snd`**, **`.eps`**, and **`.tiff`**

files in **English.lproj**. No code changes are required to make these files localizable.

- Pasteboard

  The Pasteboard method **declareTypes:num:owner:** has been changed to return an **int** instead of always returning **self**. The **int** returned is the new **changeCount** resulting from the declaration. This fixes the race condition inherent in declaring types and then asking for the **changeCount**.

- Font

  The Font class no longer gives errors when sent **alloc** or **allocFromZone**. It imposes its own zone on all objects it allocates.

- View

  Under 2.1, the **shouldDrawColor** method of View always returns YES when printing. Under 3.0, this method queries NXPrinter object when printing and might return YES or NO depending on the current printer's ability to print color.

  The View printing machinery now uses the new PrintInfo to generate print jobs. It puts comments into the Print stream based on the job features in PrintInfo. It uses

reversePageOrder to set up the generated page order of the job. In the absence of any information about the output device, it now generates pages first-to-last, rather than last-to-first.

- Timed entries

  In 2.x, timed entries were accurate to only 15 ms. In 3.0, they are accurate to ~1 ms. In addition, the time spent in the client supplied procedure used to get accumulated into the period, e.g., a timed entry with period 2 seconds that took 1 second to execute had an effective period of 3 seconds. In 3.0, the system tries to call the proc with the requested periodicity, regardless of how long the proc takes to execute. However, if the proc takes longer than the period the execute, the timed entries will not try to "catch up" to make up for the missed call(s).

Other Changes

Additional changes made to the AppKit since Release 2.1.

- If the printer is capable of accepting binary images the kit will generate images in binary format when printing. This speeds printing time by spooling smaller files.

- **NXColorSpace** enum has been extended to include **NX_CustomColorSpace**. This value indicates that the image data is to be interpreted according to the color

space in the PostScript graphics state at the time the imaging takes place and allows for imaging using custom color spaces. Note that although NXBitmapImageRep class can accept this value as the colorSpace argument, it can't write or read such images to TIFF files.

- NXMeasurementUnit enum has been added to provide a standard set of measurement units to be used in kit API. The **`NXMeasurementUnit`** default can be used to determine default setting of units in objects that show units (such as PageLayout).

- It's now possible to run an application with the global window depth limit set to a value greater than than of the default for the system. This feature should allow testing applications as if they were running on machines with deeper screens and should catch some performance problems not apparent when working just on monochorme systems. To enable this feature, use the **`NXWindowDepthLimit`** default as before, however, prefix the depth limit with "**`Test`**" to indicate that the depth limit should be set to the provided value. (Without this prefix, the depth limit is set to the minimum of the provided value and the system default.) An example:

```
dwrite appname NXWindowDepthLimit TestTwentyFourBitRGB
```

This default is for debugging and performance analysis only and should not be used under normal circumstances.

- As mentioned before, the kit will assure that application windows are resized and/or moved to properly fit screens of different sizes. Windows created in Interface Builder will also be placed on screens according to the window placement springs specified in the size inspector panel for the window. These features become especially important when running on a small screen; otherwise the title or resize bars of large windows might end up inaccessible to the user.

  To assure that an application runs properly on small screen, without actually having a small screen, you can trick the machine into believing it has one. To do this you need to edit the screen configuration stored in NetInfo. First run NetInfoManager (in /NextAdmin); you should get a window for your local host. Open the directory **`/localconfig/screens/MegaPixel`** and change the value of the **`active`** property to "1" and the width and height in the **`bounds`** property to anything you wish. Leave the origin at 0, 0. ("0 832 0 624" is one possibility; it describes a 832 x 624 screen). Then save the directory, logout, and restart the window server by specifying "exit" as the user name at the loginwindow. The system should come back up and use a smaller portion of the display as the screen.

  To restore your original screen, change **`active`** to "0" and the **`bounds`** to "0 1120 0 832" (or whatever it was initially).
- The Application class's version number has been incremented from one to two. This will serve as the version of the kit as a whole.