3.3 Release Notes: Object Links

There are no new Object Link features or bug fixes in Release 3.3, 3.2, or 3.1. This file contains release notes for Release 3.0.

Notes Specific to Release 3.0

NEXTSTEP 3.0 supports the dynamic sharing of data between documents owned by different applications via Object Links. As a typical example, imagine a user copies an illustration from Draw (the "source" application) and pastes it into a report being written in Edit (the "destination" application). If the user chooses this data to be linked, then later changes in the source drawing are automatically reflected in the report. Object Links free the user from the tedious and error prone task of manually updating dependant documents when original sources change, and allow users to share data between documents. Object Links also support hypertext-style, navigational links between documents.

In 3.0, the bundled applications that support Object Links are Draw (both as a source and destination), Edit (destination only) and IconBuilder (source only). Since Draw

is an example as well as a demo, the source for its Object Links support is available, mostly contained in the file gvLinks.m.

New Classes and Architecture

Applications participate in Object Links using the classes NXDataLinkManager and NXDataLink. A DataLink Manager coordinates link operations for a particular document. A DataLink represents a particular link between documents.

For each document that an application opens, it creates a DataLink Manager for that document. The DataLink Manager has a delegate, which is set to a custom object within the application. The DataLink Manager and its delegate cooperate to let others link to their document, to create new links to other documents and to manage existing links. The DataLink Manager does most of the real work of managing links, including storing the links and the interprocess communication used to implement links. The delegate is called on for various application-specific operations, e.g., incorporating new data for a link into a destination document.

paste.eps ¬

A key component of Object Links is a way for the system and other applications to refer to parts of a given document. A reference to a piece of a document is held in a NXSelection object. As part of creating a link to a document, an application provides

a Selection object that represents the current selection. Later the DataLink Manager may ask the delegate for a copy of the data for that selection as part of a link update. The creator of a Selection provides an arbitrary number of bytes which identify the part of the document. Since a Selection may be held by the system or other applications, its data can't be updated over time. A Selection must remain a valid reference to its part of the document, even if the document is edited in the meantime. Deciding how to represent selections of a document's data is the most difficult part of supporting Object Links, and will be elaborated on further.

A fourth class used to implement Object Links is NXDataLinkPanel (titled "Link Inspector" in the user interface). This subclass of Panel provides a standard user interface for inspecting links.

The Pasteboard class is used as the data transfer mechanism of Object Links. Pasteboards help make it easy for an application to participate in Object Links, since data transfer is implemented with existing copy and paste code. Because the owner of a Pasteboard can be lazy in providing the actual data for the types it declares, applications can freely declare as many types as they can support, yet only do the work to produce data for the types that are requested by the receiver of the data.

Creating a DataLink Manager

Before doing anything with Object Links, an application must create a DataLink Manager for each of its documents. This is not expensive, and should be done

whenever a document is opened. At creation time, the delegate for the DataLink Manager is provided, which will provide the application specific behavior for links for that document. If an existing document is being opened, the path is also passed.

```
MyCustomObjectClass *delegate;

linkManager = [[NXDataLinkManager alloc]
    initWithDelegate:delegate fromFile:pathOfDocBeingOpened];
```

If the document is untitled, a DataLink Manager should still be created, but only the delegate is passed.

```
MyCustomObjectClass *delegate;

linkManager = [[NXDataLinkManager alloc] initWithDelegate:delegate];
```

Creating an Object Link via the Pasteboard

The user creates an Object Link using an extension of the Copy/Paste metaphor. Let's take the example of a Draw illustration pasted into Edit. First, the user selects some objects within Draw and does a "Copy". In addition to its normal copy operation, Draw puts a link to the data being copied in the Pasteboard. It does this by creating a Selection that refers to the data the user is copying, and a DataLink containing that Selection. Draw then writes this DataLink to the Pasteboard. Draw must also declare an additional type to the Pasteboard, NXDataLinkPboardType (the link writes its data under this type). When the link is created, Draw declares what

data types will be able to provide when data is requested for that link (see "Updating an Object Link", below). The types available for the link can be a subset of the types initially declared to the Pasteboard, but usually are the same except for the NXDataLinkPboardType.

```
Pasteboard *pboard;
NXDataLink *newLink;
NXSelection *srcSel;
const char * const *types;

/* data types we can provide */
int numTypes;

/* number of types */
const char * const *lTypes;

/* data types we can provide later for the link */
int numLTypes;

pboard = [Pasteboard new];

/* types includes NXDataLinkPboardType */
[pboard declareTypes:types num:numTypes];

/* ...write data for any types that aren't lazy... */
srcSel = /* a selection for the current selection */

/* srcSel is handed off to the link, no need to free it */
newLink = [[NXDataLink alloc]
      initLinkedToSourceSelection:srcSel
      managedBy:myDocsLinkManager
      supportingTypes:newTypes count:numTypes];
[newLink writeToPasteboard:pb];
[newLink free];
```

The user now switches to Edit to complete the link. However, instead of choosing "Paste", the user chooses "Paste and Link". Edit first executes its normal pasting code, choosing its favorite data type from the Pasteboard's types and incorporating the data into its document. To complete the linkage to the source document, it then reads the link that Draw put in the Pasteboard, and adds it to its document via the DataLink Manager. It passes a Selection that refers to the newly pasted data.

```
Pasteboard *pboard;
NXDataLink *newLink;
NXSelection *destSel;

pboard = [Pasteboard new];
/* ...choose types, do normal pasting code... */
newLink = [[NXDataLink alloc] initFromPasteboard:pboard];
destSel = /* a selection for the pasted data */
if ([myLinkManager addLink:newLink at:destSel]) {
    /* destSel is handed off to Link Manager, no need to free it */
    /* redisplay doc with newly pasted data */
} else {
    /* link failed to be added, Link Manager puts up an Alert */
    /* rip out pasted data, since the operation failed */
[newLink free];
[destSel free];
}
```

It is quite possible for the **addLink:at:** method to fail, in which case the entire "Paste and Link" command should fail (the result should not be an unlinked paste of data, as the user can get this by doing "Paste"). Instead of the above code, it may be easier for some applications to not incorporate the data from the Pasteboard into their

document until they know the link has been successfully added to the document. This technique works as long as you can still generate the destination Selection before attempting to add the link.

Creating an Object Link to a File

In NEXTSTEP the preferred user interface for importing a file into a document is to allow the user to just drag and drop the file from Workspace into a document window. In addition, the application can let the user link a file into his document instead of copying it (the user performs the drag and drop with the control key down, just as links are made within Workspace ± see the AppKit release notes on the new image dragging support). In this case, the application imports the file's data as usual, and then adds a DataLink pointing to the imported file to its document's DataLink Manager.

```
char *filename = /* file to link to, often from dragging pboard */;
Pasteboard *pboard;
NXDataLink *newLink;
NXSelection *destSel;

newLink = [[NXDataLink alloc] initLinkedToFile:filename];

/* sel is handed off to Link Manager, no need to free */
destSel = /* a selection for the imported data */
if ([myLinkManager addLink:newLink at:destSel]) {
    /* destSel is handed off to Link Manager, no need to free it */
    /* import the file's data and redisplay */
} else {
```

```
        /* link failed to be added, Link Manager puts up an Alert */
        /* don't import the file's data, since the operation failed */
    [newLink free];
    [destSel free];
    }
```

Updating an Object Link

An Object Link update is the process of sending a new version of the linked data from the source to the destination. A number of events can cause a link to be updated: the system may notice that source document has changed, the user may explicitly request an update, or an application may programmatically initiate an update.

An update begins with the system making sure the source document is open, launching the responsible application if necessary. The source document's DataLink Manager then asks the delegate to copy a certain part of the document to a Pasteboard. It indicates the part of the document by passing the Selection that the delegate previously generated as part of the "Copy" when the link was created. Note that since the Pasteboard works lazily, the delegate doesn't have to actually produce any data at this time, but just declare the types it can provide to the Pasteboard. A further optimization is may be taken if the "cheap copy" flag is set. Normally an application that provides data to the Pasteboard lazily must put at least one representation of the data into the Pasteboard at copy time, so that when other representations are requested it has the original data to convert from. The application can't use data in the document itself, since this may be edited by the user

in the interim between the copy and the paste. If cheap copy is allowed, then the application does not need to write any representations at copy time, because it is guaranteed that there will be no events processed before the owner is asked to provide data for the types it declares (for example, Draw uses the optimization to avoid archiving the objects in the selection for later conversion to EPS or TIFF, and instead produces the EPS or TIFF directly from the object in the document ± see Draw's source code).

```
- copyToPasteboard:(Pasteboard *)pboard at:(NXSelection *)selection
      cheapCopyAllowed:(BOOL)flag
{
     /* ...copy new data into pboard... */
}
```

The update continues as the new data is transferred into the destination document. The document's DataLink Manager asks its delegate to paste the contents of the Pasteboard previously set up by the source, passing a Selection to indicate the location.

```
- pasteFrom:(Pasteboard *)pboard at:(NXSelection *)selection
{
     /* ...paste new data from pboard... */
}
```

If the link was created to a file instead of via the Pasteboard, then a source application is not involved, and the delegate of the destination is simply told to import new data from that file.

```
- importFile:(const char *)filename at:(NXSelection *)selection
{
    /* ...import new data from file... */
}
```

Copying and Pasting Data Containing Object Links

What happens if the user copies some data that is already linked?

When copying data between applications, since none of the standard Pasteboard types are rich enough to describe what components are linked and how they can be updated, links are usually not transferred. For example, imagine a user creates a graph in a charting application and pastes it, with linking, into Draw for annotation. The user then scales the whole graph down, and adds some titles and call-outs. He then selects the entire result in Draw, and does a "Copy".

1) If he then goes to Edit and does a "Paste", an EPS representation of the data will be pasted into the Edit document, but there will be no link made in the Edit document. Changes to the chart will be propagated to the Draw document, but the user will have to manually re-copy the data into Edit to have the new chart appear in Edit.

2) If the user does a "Paste and Link", a link will be made from Edit to Draw. Changes in the graph will propagate to Draw, and these changes will then propagate to Edit. Note that the linking from the Edit document to the graph

must go through Draw, because only Draw knows how to scale the graph and add its call-outs to create the final illustration.

When copying data within a single application, representations of the data that are richer than the standard Pasteboard types are usually transferred. This is also the case when copying linked items within an application. In this case, links should be preserved. For example, imagine a user has some RTF in Edit containing a linked illustration from Draw. She selects a range of text that includes the graphic.

3) If she then goes to another Edit document, or a different place in the same document, and does a "Paste", a copy of the text and the graphic is pasted, and the new graphic is still linked. This behavior also implies that a linked graphic can be cut from one part of a document and pasted into another part without the graphic becoming non-linked.

4) If she does a "Paste and Link", a link is made to the material that was copied. In this case the "Paste and Link" must be done in a different document than the copy (circular links are not allowed). Doing "Paste and Link" always creates a new link to the place where the last "Copy" was done. (This example is actually fictitious since Edit does not support being a link source, so it cannot be linked to.)

To implement case 3) above, if an application is copying data that contains linked items, it should message the DataLink Manager to write the link data for the document to the Pasteboard along with the private data it normally copies. It should not try to write the DataLinks themselves along with its private data. When the

private data is pasted, this link information will be used to create new links in the pasted data.

```
[myLinkManager writeLinksToPasteboard:pboard];
```

When the user does a "Paste" and the private data type is being inserted into the destination, as the paster encounters the previously linked items embedded in the private data it asks its DataLink Manager to incorporate each incoming link from the pasteboard into the receiving document. At this point, a DataLink is created and returned.

```
Pasteboard *pboard;
NXDataLink *newLink;
NXSelection *oldDestSel;
NXSelection *newDestSel;

pboard = [Pasteboard new];

/* while pasting private data from board, the following is done
    for each linked item encountered in the data being pasted
*/

oldDestSel = /* the selection for the link in the old document */;
newDestSel = /* a selection for the newly pasted data */
newLink = [myLinkManager addLinkPreviouslyAt:oldDestSel
                                fromPasteboard:pboard at:newDestSel];
if (newLink) {
     /* newDestSel is handed off to Link Manager, no need to free */
     [oldDestSel free];
} else {
     /* paste fails */
     [newDestSel free];
```

```
        [oldDestSel free];
    }
```

Keeping the DataLink Manager Informed

In order for the DataLink Manager to keep linked data up to date, it must be kept apprised of the state of the document. The delegate fulfills this duty by sending the DataLink Manager messages when various commands are applied to the document, such as "Save", "Save As", "Save To", "Revert to Saved" and "Close". The delegate also sends a message to the DataLink Manager whenever any part of the document is edited. The messages are

- documentSaved;
- documentReverted;
- documentSavedAs:(const char *)path;
- documentSavedTo:(const char *)path;
- documentClosed;
- documentEdited;

Learning that the Document's Link Data has been Edited

Since the data kept for Object Links is part of the destination document, the document should be considered edited when this information is changed. The link data is edited when the user changes the update mode of a link, or when the information used to find the source files of links is changed. The delegate should mark the document as edited in its implementation of the method

```
- dataLinkManagerDidEditLinks:(NXDataLinkManager *)sender;
```

Creating Selections

In order for an application's documents to be sources of linked data, it must be able to produce Selection objects which represent parts of a document. A Selection contains an application determined array of bytes of any length (the Selection's "description"). Selections are persistent, immutable references. It is desirable that Selections be maintainable without having to store state in the document to whose parts they refer. This allows users to link to documents which they have permission to read but not modify.

For example, in a drawing program, each figure created in a document could be assigned an incrementing integer. A given selection of objects would then be represented as the list of their integers. Or if the user selects an area of the illustration, the rectangle selected would represent the selection. It is common to link to the entire contents of a document (e.g., a whole graph linked into a report, or a whole text file linked into a page layout). In this case, a special selection can created that means "the same selection that results from the user performing 'Select All'" by using the **allSelection** factory method of NXSelection. All these types of Selections will hold their meaning even if the illustration is edited, and do not require writing to the source document.

It is more difficult some applications to produce Selections that can withstand edits to

the document while still not writing to the document. For example, linking to an arbitrary range of text usually requires that the range is recorded with the document, so that subsequent edits can keep it current. In some applications, it may be possible to link to certain selections only if the document can be modified, while simpler selections do not require saving state in the document. For example, it should be possible to link to the entire contents of a text file without having to record state in the document. Even linking to whole paragraphs should also be possible, since they could each be given a unique ID.

When an application is about to generate a Selection for which it will need to store state in the document to maintain the Selection, it should be sure it has permission to write to the document.

If an application needs to store state in its document when it generates a selection, there is also an issue of when this state is reaped. If an application is generating this state on every "Copy" command, most of this information will become garbage because creating a link is much rarer than simply pasting data. To find out whether the link put in the Pasteboard is actually used, the app can implement the method **startTrackingLink:** to find out when a link is actually added to a destination document (see "Update Modes for Object Links", below). The application can also be notified when it loses ownership of the Pasteboard by having the owner declared to the Pasteboard implement the method **pasteboardChangedOwner:**. At this point, it can know not to retain the state it kept in the document for that link. Once the link is completed, and the selection state is committed to the source document, there is currently no way to know a safe time to reap this state, because destinations

of this link can be copied and stored on removable media.

Applications that are both link sources and destinations will sometimes use different types of Selection descriptions in these scenarios. Since a user's selection may be quite complex, but the destination of a link is usually a single graphic, source Selections are often more complex. One important difference is that the API requires an application to resolve source Selections (e.g., finding the source data during an update), but never to exactly regenerate them. There may be more than one Selection that validly refers to a given user selection. On the other hand, the API does require that an application regenerate destination Selections. This is used when previously linked data is being pasted to refer to the links in the copying document. Destination Selections are also used as the keys for finding a particular link in a destination document. For example, when a linked item is deleted, the DataLink Manager's delegate must break that link, and it looks up that link by its destination Selection.

Sometimes if an application uses different strategies for Selections descriptions, one of the first bytes can be used to indicate the type of Selection. This byte can also be used for versioning.

Selection descriptions should also be architecture independent. One way to do this is to use an ASCII string. Another is to use the the first byte to indicate the byte-sex of the data.

Storing Object Link Information

Information about Object Links persists only in the destination document. This allows read-only documents to be linked to. How does the DataLink Manager store this information along with the document?

Documents that hold links to other documents are required to be file packages (i.e., the document is a directory instead of a single file). The DataLink Manager is then able to store its information in a file within the package without interfering with the storage of the rest of the document. This solution also has the advantage that the system can see what links a document has without having to launch the application to parse this information out of its private document format.

Update Modes for Object Links

Object Links have three different update modes, which determine how often a link is updated. In manual mode, links are updated only when the user explicitly requests an update. In when-source-saved mode, links are updated when the system determines that the source has changed since the last update. Such updates can occur when documents are opened or saved. In continuous mode, links are updated whenever the source data is changed. The update mode of a link can be set by the user in the Link Inspector. The initial mode of a new link is controlled by the default "NXDataLinkUpdateMode", which may be set globally for all applications or specifically for a given application.

A DataLink Manager delegate must do a little extra work to support continuous updating; otherwise this option will not be available to the user. The delegate must be able to tell the DataLink Manager whenever the sources of links are edited. It must first respond YES to the message

```
- (BOOL)dataLinkManagerTracksLinksIndividually;
```

How does it know what parts of its document it should be tracking as sources of links? As the DataLink Manager for the source document learns of dependant documents that are currently open, it learns of selections within the source document that are linked to. The delegate can be notified as these associations are made by implementing

```
- dataLinkManager:(NXDataLinkManager *)sender
      startTrackingLink:(NXDataLink *)link;
- dataLinkManager:(NXDataLinkManager *)sender
      stopTrackingLink:(NXDataLink *)link;
```

The delegate asks these links for their source selection, and then tracks changes to these selections. When one of these pieces of the document is edited, it messages the link corresponding to that selection.

```
[link sourceEdited];              /* after link's source data is edited */
```

These messages drive the continuous updates. Implementing this fine grained tracking of changes also allows the DataLink Manager to take some optimizations

when updating automatic links during a save of the source document, since it knows specifically which dependants need to be updated.

Link Buttons

Link Buttons are diamond shaped buttons that take the user to a place in another document when clicked. They are used in the Help System and Object Links.

The user creates a Link Button by starting in the source document and doing a "Copy". In the destination, she then chooses "Paste Link Button". The destination's implementation is similar to that of "Paste and Link", except simpler since no data is transferred with Link Buttons.

```
Pasteboard *pboard;
NXDataLink *newLink;
NXSelection *destSel;

pboard = [Pasteboard new];
newLink = [[NXDataLink alloc] initFromPasteboard:pboard];
destSel = /* a selection for the pasted data */
if ([myLinkManager addLinkAsMarker:newLink at:destSel]) {
    /* destSel is handed off to Link Manager, no need to free it */
    /* import the Link Button image, redisplay doc */
} else {
    /* link failed to be added, Link Manager puts up an Alert */
    [newLink free];
    [destSel free];
}
```

There are named NXImages for the Link Button ("NXLinkButton") and the highlighted Link Button ("NXLinkButtonH"). By default these images are about 12 units high. If you wish to use larger buttons, or if your application allows the user to scale them, you may create larger images by first copying the named NXImage and then changing your copy's size. Don't try to scale the named NXImage, since other objects in the system will be affected. Draw demonstrates this in the source file Image.m. (Note: In the 3.0 Beta Release, Draw has some vestigial tiff images for links in its project, which it does not even use in its code. Be sure to use the named images for Link Buttons provided by the AppKit).

When the user clicks on a link button, applications should track the mouse as they would for any button, so the user can move the mouse in and out of the button and see feedback in the way the button is highlighted. It is acceptable to track the rectangle enclosing the button instead of the diamond shape itself. For Link Buttons in a text streams, this means that clicking in a Link Button and dragging should not initiate text selection tracking. For Link Buttons in drawing applications, clicking one the button and moving may collide with the gesture used to move the Link Button graphic. We recommend that you treat the button as a button if it is not a selected graphic, and treat it as a normal graphic if it is selected. In addition, if the graphic is "locked", then it can always be treated as a button (thus authors can lock the Link Buttons so they function unambiguously for their end users). Draw demonstrates this behavior.

Showing Destination Object Links

Applications that serve as link destinations should allow the user to see which items are links. The DataLink Manager delegate should implement the method

```
- dataLinkManagerRedrawLinkOutlines:(NXDataLinkManager *)sender;
```

When it receives this method, it should redraw the linked items currently in view. Whenever linked items are drawn, they should ask their DataLink Manager whether links are currently being show with the method

```
- (BOOL)areLinkOutlinesVisible;
```

If links are visible, they should be drawn with link outlining, using the C function

```
void NXFrameLinkRect(const NXRect *aRect, BOOL isDestination);
```

This function draws a distinctive link outline just outside the provided rectangle. For destination links **isDestination** should always be YES. For erasure or other purposes, the thickness of this outline can be found be calling **NXLinkFrameThickness()**.

Do not draw link outlines around Link Buttons. They are already clearly identifiable as links, and the border around them is superfluous.

Showing Source Object Links

In applications that serve as link sources, the DataLink Manager delegate should implement two simple methods to allow the Open Source command to function properly.

```
- windowForSelection:(NXSelection *)selection;
- showSelection:(NXSelection *)selection;
```

The first of these simple returns the Window object that is displaying the given selection. Since most documents only have one window, this should be trivial for the delegate to implement. The second method is used during open source to show the user the source of the link. This method should scroll into view the data references by the given selection. It may also draw link borders around the source data using **NXFrameLinkRect()** with **isDestination** passed as NO.

The Link Inspector

The Link Inspector lets the user perform a few basic operations on the Object Links in a document. Since linking state is only stored in the destination document, this panel is only available to inspect links within the destination application. The panel allows the user to manually update a link, open the source of a link, break a link (i.e., prevent any future updates), break all links and set the update mode of a link.

The menu item that brings up the panel should send the **orderFrontDataLinkPanel:** message to NXApp.

Similar to the Font Panel, the DataLink Panel must track the user's selection as she selects data that is linked or contains links. It is the application's responsibility to send the panel a message as the selection changes. Be sure to keep the panel up to date when documents are opened, become key and are closed. Since this state needs to be tracked even before the panel is created, these messages can be sent to the factory as well as instances of the panel.

```
[NXDataLinkPanel setLink:currentlySelectedLink
    andManager:myLinkManager
    isMultiple:flag];
```

Breaking and Deleting Object Links

When the user deletes a linked item, the DataLink Manager's delegate should inform the manager that the link is no longer part of the document. It can do this by simply sending the appropriate DataLink the **break** method. After the link is no longer part of the destination DataLink Manager, it may be freed.

```
NXDataLink *linkToBreak;
NXSelection *destSel;

destSel = /* a selection for the data begin deleted */
linkToBreak = [myLinkManager findDestinationLinkWithSelection:destSel];
if (linkToBreak) {
    [linkToBreak break];
    [linkToBreak free];
}
```

When a link is broken, the DataLink Manager sends its delegate the message

```
- dataLinkManager:(NXDataLinkManager *)sender
    didBreakLink:(NXDataLink *)link;
```

The delegate will probably wish to note that the graphic in question is no longer a linked item, and redraw without link borders if links are being shown. If the link transfers no data (i.e., its a Link Button), the button should be deleted, since it has no reason to exist as a broken link.

Publishing an Object Link

Sometimes it is convenient for users to be able to save an Object Link in the file system, instead of moving the link between documents via the Pasteboard. For example, there may be a specific picture or range of spreadsheet cells that a document owner may want others to link to. The user does this with the "Publish Selection" command. The DataLink class' **saveLinkIn:** method supports this feature.

```
NXDataLink *newLink;
NXSelection *srcSel;
char *directoryName;      /* the directory of the doc we are publishing from */
const char * const *lTypes;
/* data types we can provide later for the link */
int numLTypes;
```

```
srcSel = /* a selection for the current selection */
/* srcSel is handed off to the link, no need to free it */
newLink = [[NXDataLink alloc]
      initLinkedToSourceSelection:srcSel
      managedBy:myDocsLinkManager
      supportingTypes:lTypes count:numLTypes];
[newLink saveLinkIn:directoryName];
[newLink free];
```

On the destination side, the link is imported by the user by dragging it into the destination document. The destination can recognize a published link being imported by the ".objlink" suffix. Handling a published link is very similar to implementing "Paste and Link". The destination application simply instantiates a DataLink from the file, and adds it to its DataLink Manager.

```
char *filename;              /* name of file being imported */
NXDataLink *newLink;
NXSelection *destSel;

newLink = [[NXDataLink alloc] initFromFile: filename];
destSel = /* a selection for the pasted data */
if ([myLinkManager addLink:newLink at:destSel]) {
      /* destSel is handed off to Link Manager, no need to free it */
      [newLink update];      /* get initial copy of the data */
} else {
      /* link failed to be added, Link Manager puts up an Alert */
      [newLink free];
      [destSel free];
}
```

Object Links Menu Items

The recommended menu structure for Object Links starts a submenu of the "Edit" menu titled "Link". This item should come immediately after the "Paste" command and any variants of "Paste". The items in the "Link" menu should be, in order,

Paste and Link
Paste Link Button
Publish Selection...
Show Links
Link Inspector...

"Paste and Link" has an optional command key equivalent of "V". Additional menu items may be added after "Publish Selection...". "Show Links" should toggle to "Hide Links" when links are being shown already for the document of the key window. All these menu items and their behavior can be seen in Draw.

Control double-clicking on a linked item is the recommended accelerator for doing an "Open Source" on that link. This is implemented by looking up the DataLink for that link and sending it the **openSource** message.

Proper Launch Behavior

When an application is launched to open the Source of an Object Link it is not appropriate for the application to create a new document, as it might otherwise do on

launch. The application may detect this case by checking for the NXServiceLaunch default (which is also set when the app is launched to provide a service). Often this check is made in the **appDidInit:** method of NXApp's delegate.

```
if (!NXGetDefaultValue([self appName], "NXServiceLaunch"))
    /* make a new document on launch */
```

Choices when Control-Dragging files

When the user control-drags a file into a document window, there are three different options available to the application:

1) Import the file's data, and create an Object Link to that file (this is only possible of the file's data is of a type that can be imported).

2) Show a Workspace-style icon for the file, which serves as a navigational link to the file.

3) Show a Link Button, which serves as a navigational link to the file.

If your app supports two or more of these options, the recommended way to make this decision is to put up an Alert prompting the user to make a choice. See Draw for an example of this behavior. Note that Draw puts up this Alert in its **concludeDragOperation:** method, because the earlier drag methods are sensitive to timeouts and the user may take an arbitrarily long time to dismiss the Alert.