Chapter 4

Converting Other Kits

This chapter describes conversions for kits other than the Application Kit and the Common classes. Many of the conversions described here are optional in that you don't have to run these scripts unless your program uses classes from the kit. (To learn how to run an optional conversion script, see the on-line document *Converting Your Code to OpenStep* in **/NextLibrary/Documentation/NextDev/ReleaseNotes**.) Not all NEXTSTEP kits have a conversion script.

Custom IB API Conversion

Optional

If you are converting a palette project, you must run the script **CustomIBAPI.tops** to convert public Interface Builder API. Like the API declared in the Application Kit, the public API for Interface Builder is essentially the same, but it has been modified to use the Foundation Framework and OpenStep programming conventions.

Note: It's probably best to perform a deep conversion if you're converting a palette project. Otherwise, the conversion process won't catch things like List instance variables that are now NSArrays.

Changes to Protocols

If you've written a palette, you know that much of the Interface Builder API is declared in protocols that you implement to accomplish certain tasks. Because of the extensive use of protocols, it's tricker to catch errors in

the conversion process. Many of the methods declared in protocols have had name changes, but if you still implement the old version of the method, the compiler does not understand that you meant to implement a method from the protocol, and it won't flag it as an error. For this reason, you'll have to look especially closely at your use and implementation of Interface Builder protocol methods. The most problematic protocols are listed here. You should verify that you're implementing the new version of each method in the protocol.

IBDocuments

The following table lists each of the methods in the IBDocuments protocol, that method's new name, and any differences in the arguments or return values. If you implement the IBDocuments protocol, make sure you're implementing the new versions of the methods. You may also need to change messages that you send to **[NSApp activeDocument]**.

Old Method	New Method	Differences
TableHeadRule.eps ¬		
addConnector: TableRule.eps ¬	addConnector:	Returns void.
attachObject:to: TableRule.eps ¬	attachObject:toParent:	Returns void.
attachObjects:to: TableRule.eps ¬	attachObjects:toParent:	Objects list is NSArray. Returns void.
copyObject:type:inPasteboard: TableRule.eps ¬	copyObject:type:toPasteboard:	Pasteboard type is NSString. Returns BOOL.
copyObjects:type:inPasteboard: TableRule.eps ¬	copyObjects:type:toPasteboard:	Objects list is NSArray. Pasteboard type is NSString. Returns BOOL.
deleteObject: TableRule.eps ¬	detachObject:	Returns void.
deleteObjects: TableRule.eps ¬	detachObjects:	Objects list is NSArray. Returns void.
editorDidClose:for: TableRule.eps ¬	editor:didCloseForObject:	Returns void.

getDocumentPathIn: TableRule.eps ¬	documentPath	Returns path as NSString.
getEditor:for: TableRule.eps ¬	editorForObject:create:	Arguments are reversed.
getNameIn: TableRule.eps ¬	nameForObject:	Returns name as NSString.
getObjects: TableRule.eps ¬	objects	Returns object list as NSArray.
getParentForObject: TableRule.eps ¬	parentForObject:	
listConnectorsForDestionation: TableRule.eps ¬	connectorsForDestination:	Returns connector list as NSArray.
listConnectorsForDestionation:filterClass: TableRule.eps ¬	connectorsForDestination:ofClass:	Returns connector list as NSArray.
listConnectorsForSource: TableRule.eps ¬	connectorsForDestination:	Returns connector list as NSArray.
listConnectorsForSource:filterClass: TableRule.eps ¬	connectorsForDestination:ofClass:	Returns connector list as NSArray.
objectIsMember: TableRule.eps ¬	containsObject:	
openEditorFor: TableRule.eps ¬	openEditorForObject:	
removeConnector: TableRule.eps ¬	removeConnector:	Returns void.
pasteType:fromPasteboard:parent: TableRule.eps ¬	pasteType:fromPasteboard:parent:	Returns NSArray of objects. Pasteboard type is NSString.
redrawObject: TableRule.eps ¬	drawObject:	Returns void.
setName:for: TableRule.eps ¬	setName:forObject:	Name is an NSString.
setSelectionFrom:	setSelectionFromEditor:	Returns void.

TableRule.eps ¬		
touch	touch	Returns void.
TableRule.eps ¬		

IBEditors

The following table lists the each changed method from the IBEditors protocol, its new name, and any differences in the arguments or returns values for the new method.

Old Method	**New Method**	**Differences**
TableHeadRule.eps ¬		
acceptsTypeFrom:	acceptsTypeFromArray:	Returns BOOL. Type list is NSArray of NSStrings.
TableRule.eps ¬		
close	close	Returns void. Must release editor in this method.
TableRule.eps ¬		
closeSubeditors	closeSubeditors	Returns void.
TableRule.eps ¬		
makeSelectionVisible:	makeSelectionVisible:	Returns void.
TableRule.eps ¬		
orderFront	orderFront	Returns void.
TableRule.eps ¬		
resetObject:	resetObject:	Returns void.
TableRule.eps ¬		
selectObjects:	selectObjects:	Returns void. Object list is NSArray.
TableRule.eps ¬		

IBSelectionOwners

IBSelectionOwners protocol has changed as summarized in the following table.

Old Method TableHeadRule.eps ¬	New Method	Differences
selectionCount TableRule.eps ¬	selectionCount	No change
getSelectionInto: TableRule.eps ¬	selection	Returns NSArray.
redrawSelection TableRule.eps ¬	drawSelection	Returns void.

IBConnectors

The changes to the IBConnectors protocol are very slight. However, there is one new method declared in this protocol that you should implement:

- (NSString *)**label**

This method is used to label the connection when it is displayed in the nib file window in outline mode.

IBInspectors

The IBInspectors protocol is obsolete. Its methods are now part of the IBInspector class. If you have a class that specifically adopts the IBInspectors protocol, you can remove this from the declaration.

IBDocumentControllers

The IBDocumentControllers protocol is obsolete because its functionality is implemented through

notifications. Interface Builder's application object posts the following notifications under the following circumstances, rather than sending messages directly to registered objects. If you implement the IBDocumentControllers protocol, you should add your object as an observer of these notifications. (See ªNotification Conversionº in ªConverting Other Kitsº for a detailed description of notifications.)

Notification TableHeadRule.eps ¬	Posted When
IBDidOpenDocumentNotification TableRule.eps ¬	A document has been opened.
IBDidSaveDocumentNotification TableRule.eps ¬	A document has been saved.
IBWillCloseDocumentNotification TableRule.eps ¬	A document is about to be closed.
IBWillSaveDocumentNotification TableRule.eps ¬	A document is about to be saved.

Obsolete Method TableHeadRule.eps ¬	Replacement
didOpenDocument: TableRule.eps ¬	Register to receive IBDidOpenDocumentNotification.
didSaveDocument: TableRule.eps ¬	Register to receive IBDidSaveDocumentNotification.
willSaveDocument: TableRule.eps ¬	Register to receive IBWillSaveDocumentNotification.
registerController: TableRule.eps ¬	Register to receive individual notifications.
unregisterController: TableRule.eps ¬	[[NSNotificationCenter defaultCenter] removeObserver:]

Old Code

```
@interface MyController : NSObject <IBDocumentControllers>
...
@end

@implementation MyController
- init
{
     ...
     [NXApp registerController:self];
}

- didOpenDocument:theDocument {...}
- didSaveDocument:theDocument {...}
- willSaveDocument:theDocument {... }

- free
{
     [NXApp unregisterController:self];
     ...
}
...
@end
```

New Code

```
@interface MyController : NSObject
```

```
...
@end

@implementation MyController
- init
{
     ...
     [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(didOpenDocument:)
        name:IBDidOpenDocumentNotification object:NSApp];
     /* also add yourself as an observer of
        IBDidSaveDocumentNotification and
        IBWillSaveDocumentNotification
     */
}
- (void)didOpenDocument:(NSNotification *)theNotification
{
     IBDocument *theDocument = [theNotification object];
     ...
}
- (void)didSaveDocument:(NSNotification *)theNotification {...}
- (void)willSaveDocument:(NSNotification *)theNotification {...}
- (void)dealloc
{
     [[NSNotificationCenter defaultCenter] removeObserver:self];
     ...
}
...
@end
```

Changes to View Additions

If you implement **getMinSize:maxSize:from:**, you should remove that method from your code. This method has been replaced by two methods: **minimumSizeFromKnobPosition:** and **maximumSizeFromKnobPosition:**. Both of these methods take an argument of type **IBKnobPosition**, which is an enumerated type. All of the knob position constants are now part of the **IBKnobPosition** enum.

In addition, the category on NSView now declares the method **allowsAltDragging**. Implement **allowsAltDragging** and have it return YES if your NSView subclass can be Alternate-dragged to create a matrix.

Also, there is now a category on NSCell that allows you to control resizing and Alternate-dragging of cells as well as views.The NSCell additions are declared in the header file **NSViewAdditions.h**.

More Additions to NSObject

The category on NSObject has the following new method:

- (NSString *)**classInspectorClassName**

This method returns the class name for the custom class inspector. The custom class inspector is new in this release of Interface Builder. (It looks like the attributes inspector for custom objects.) Using the custom class inspector, you can create subclasses of NSView subclasses like NSButton without losing the attributes inspector for the button. The default custom class inspector should be sufficient in most cases.

An additional category on NSObject declares this method:

+ (BOOL)**canSubstituteForClass:**(Class)*originalObjectClass*

You implement this method to have your class not be displayed in the Custom Class inspector of its superclass. This is necessary in rare instances only. For example, NSForm is a subclass of NSMatrix, but it is so specialized that it wouldn't be appropriate for a programmer to create an NSMatrix and assign it the NSForm class. Thus, NSForm overrides **+canSubstituteForClass:** as shown.

New Code

```
@implementation NSForm (IBCustomClassInspector)

+ (BOOL)canSubstituteForClass:(Class)originalObjectClass
{
     return (originalObjectClass == [NSMatrix class]) ? NO : YES;
}
```

New Notifications

In addition to the notifications that replace the IBDocumentControllers protocol, the new Interface Builder API now uses these notifications:

Notification TableHeadRule.eps ¬	Posted when
IBWillAddConnectorNotification TableRule.eps ¬	A connection is about to be made.
IBDidAddConnectorNotification TableRule.eps ¬	A connection has been made.
IBWillRemoveConnectorNotification TableRule.eps ¬	A connection is about to be removed.
IBDidRemoveConnectorNotification TableRule.eps ¬	A connection has been removed.
IBSelectionChangedNotification TableRule.eps ¬	The selection has changed.
IBInspectorDidModifyObjectNotification TableRule.eps ¬	The user has changed values on the inspector.

Obsolete Methods

The header file **InterfaceBuilder.h** used to declare methods that were used in previously releases of Interface Builder, but have been obsolete since release 3. **InterfaceBuilder.h** no longer declares these methods, and you should no longer use them in your code.

Obsolete Method	**Replacement**
TableHeadRule.eps ¬	
disableUpdate (in IB protocol) TableRule.eps ¬	None
reenableUpdate (in IB protocol) TableRule.eps ¬	None
updateFor: (in IBInspector) TableRule.eps ¬	None

DO Conversion

Stage 1

As mentioned at the beginning of this guide, the Distributed Objects system is now part of the Foundation Framework. OpenStep uses the same encoding scheme to distribute objects as it does to archive them, so the NXTransport protocol is no longer necessary. The changes to the Distributed Objects API are summarized here.

Class or Protocol TableHeadRule.eps ¬	Replacement
NXConnection TableRule.eps ¬	NSConnection
NXProxy TableRule.eps ¬	NSDistantObject
NXPort TableRule.eps ¬	NSPort (not part of OpenStep)
NXTransport TableRule.eps ¬	NSCoding

Encoding for Distribution

As mentioned previously, OpenStep uses the same encoding scheme to distribute objects as it does to archive them. You should be able to implement just one **encodeWithCoder:** and one **initWithCoder:** method that both the archiving system and the distributed objects system can use.

As you'll recall, the NSCoder class, which defines the interface for objects that encode and decode objects, is an abstract class. For archiving, NSArchiver and NSUnarchiver objects do the real work. There is no OpenStep class that encodes objects for distribution, but NeXT provides the NSPortCoder class to perform this function. Where you have referenced an encoding object specifically in your code, the conversion script will change it to an instance of NSPortCoder.

Changes to NXConnection

These few changes have been made to the NXConnection class. The changes made to your code are summarized in the table below.

SquareBullet.eps ¬NSConnection doesn't support the notion of placing all connections in a zone.

SquareBullet.eps ¬There's no way to unexport an object, as with **[NXConnection removeObject]**. The proxies for connected objects go away when the connection goes away.

SquareBullet.eps ¬Timeout values are given in seconds rather than milliseconds. Input and output timeouts are now set with two different methods.

Obsolete Method TableHeadRule.eps ¬	**Possible Replacement**
setDefaultZone: TableRule.eps ¬	Use allocWithZone: to create connections
defaultZone: TableRule.eps ¬	Reference zone used to allocate the connections.
removeObject TableRule.eps ¬	None
setDefaultTimeOut: TableRule.eps ¬	setRequestTimeout: and setReplyTimeout: (instance methods)

Connecting to Ports

There are no OpenStep equivalents to the **connectToPort:...** methods; however NeXT has extended the NSConnection class description to include methods that connect using NSPort objects. The DO conversion converts your **connectToPort:...** messages to their NeXT equivalents.

Running a Connection

NXConnection had its own internal run loop to run a connection. You could send the connection **run**, **runFromAppKit** (to use the Application Kit's run loop), or **runInNewThread**. NSConnection and the Application Kit now use the same run loop. To run it, you send **[[NSRunLoop currentRunLoop] run]**. If you want to allow timers or Application Kit objects to run while the connection is waiting for a response, run the loop in

NSConnectionReplyMode:

```
[NSRunLoop currentRunLoop] runMode:NSConnectionReplyMode
€€€€€beforeDate:[NSDate distantFuture]];
```

To run the connection in a different thread, use NSThread. Create a method that performs the work you want to do in a new thread, then use **detachNewThreadSelector:toTarget:withObject:** to detach the thread.

Old Code

```
[aConnection runInNewThread];
```

New Code

```
[NSThread detachNewThreadSelector:@selector(runMe) toTarget:self
     withObject:nil];
...
- (void)runMe
{
     [[NSRunLoop currentRunLoop] run];
}
```

Retrieving the Number of Messages Received

The **messagesReceived** class method is obsolete. Each NSConnection instance keeps track of statistics about itself in an NSDictionary returned by the **statistics** method. Send the **description** message to this NSDictionary to see the type of statistics kept.

DPS Conversion

Stage 3

The extensions to the DPS client library now use objects wherever possible. Where the use of objects was not possible, the API has changed to conform more closely with the Adobe naming conventions. The new OpenStep interfaces are in the header file **dpsOpenStep.h**. It and all other DPS header files are now in **AppKit.framework**.

The biggest change is the new NSDPSContext class. You now create a DPS context by creating an NSDPSContext object. This object provides methods that perform any operation you typically perform on a context. If you are more familiar with the C function interface, you can use the method **DPSContext** to retrieve the object's DPSContext record. You can then operate on this context record using any of the functions or single operator functions defined in the DPS client library. The object and the record are always synchronized with each other; if you change one, the other is always automatically updated.

Also, to conform to the rest of OpenStep, the DPS system now uses NSRunLoop objects. Conversion to NSRunLoops requires some manual conversion if you are creating timed entries, accessing ports, or accessing file descriptors.

DPSAddTimedEntry(), DPSRemoveTimedEntry()

Timed entries are handled with NSTimer objects in OpenStep. NSTimer is a Foundation class that defines timer objects that work with NSRunLoop objects. During this conversion, the **DPSAddTimedEntry()** and **DPSRemoveTimedEntry()** functions are changed as shown in the following example.

Old Code

```
void myHandler(DPSTimedEntry teNumber, double now, void *who)
{
     [(id)who tick];
}
```

```
...
- setRefreshSpeed:(double)theSpeed
{
     refreshSpeed = theSpeed;
     if (mvFlags.running) {
         DPSRemoveTimedEntry(timedEntry);
         timedEntry = DPSAddTimedEntry(refreshSpeed, myHandler,
             self, NX_BASETHRESHOLD);
     }
     return self;
}
```

Bad New Code

```
void myHandler(NSTimer *teNumber, double now, void *who)
{
     [(id)who tick];
}
...
- setRefreshSpeed:(double)theSpeed
{
     refreshSpeed = theSpeed;
     if (mvFlags.running) {
         [timedEntry invalidate]; [timedEntry release];
         timedEntry = [[NSTimer
             scheduledTimerWithTimeInterval:refreshSpeed
             target:<target> selector:myHandler
             userInfo:self repeats:YES] retain];
     }
     return self;
}
```

The **timerWithTimeInterval:target:selector:userInfo:repeats:** method creates a timer object that repeatedly schedules itself to fire after **refreshSpeed** seconds. When the NSTimer created in this example fires, it sends the message **[<target> myHandler:self]**.

You must complete the conversion by doing the following:

SquareBullet.eps ¬Change the selector argument in **timerWithTimeInterval:target:selector:userInfo:repeats:** to a method selector. The method must return **void** and take an NSTimer object as an argument.

SquareBullet.eps ¬Change the target argument to the object that defines the method you specified for the selector argument.

SquareBullet.eps ¬If your method needs more information than just the timer object, pass that information through the userInfo argument. Then, in the method's implementation, send the message **[timer userInfo]** to extract that information from the timer.

In the above example, the **myHandler** function can be removed, the **tick** method used as the selector, and **self** used as the target. The **tick** method must be changed so that it takes an NSTimer object as an argument. The following code shows these changes.

Good New Code

```
- setRefreshSpeed:(double)theSpeed
{
     refreshSpeed = theSpeed;
     if (mvFlags.running) {
         [timedEntry invalidate]; [timedEntry release];
         timedEntry = [[NSTimer scheduledTimerWithTimeInterval:
             (NSTimeInterval)refreshSpeed
             target:self selector:@selector(tick:)
             userInfo:nil repeats:YES] retain];
```

```
        }
        return self;
    }
```

Ports and File Descriptors

The OpenStep specification does not contain API for listening to ports or file descriptors as part of the NSRunLoop because ports and file descriptors are operating-system specific. If you have code that performs these operations, it will not be portable. The conversion process changes these functions to API provided by NeXT that is not part of the OpenStep specification.

Previously, you called **DPSAddPort()** or **DPSAddFD()** to add a port or file descriptor and to register a function that would handle events involving that port or file descriptor. You used **DPSRemovePort()** and **DPSRemoveFD()** to remove the port or file descriptor. All four of these functions are now obsolete. You must now perform these steps instead:

1. Create an NSPort or NSPosixFileDescriptor object and associate it with the port or file descriptor.
2. In the case of file descriptor, have the NSPosixFileDescriptor monitor activity by invoking the method **monitorActivity:**.
3. Set a delegate for the NSPort or NSPosixFileDescriptor. The delegate will handle the events involving that port or file descriptor.
4. Register the NSPort or NSPosixFileDescriptor with the current NSRunLoop.
5. When you're done with the port or file descriptor, remove it from the NSRunLoop.

The table below summarizes the changes.

Obsolete Function TableHeadRule.eps ¬	Replacement in NeXT's implementation of OpenStep
DPSAddPort() TableRule.eps ¬	[[NSPort portWithMachPort:] retain] [*nsport* setDelegate:] [*nsrunLoop* addPort:forMode:]
DPSRemovePort() TableRule.eps ¬	[*nsrunLoop* removePort:forMode:]
DPSAddFD() TableRule.eps ¬	[[NSPosixFileDescriptor alloc] initWithFileDescriptor:] [*nsposixFileDescriptor* monitorActivity:] [*nsposixFileDescriptor* setDelegate:] [*nsrunLoop* addPosixFileDescriptor:forMode:]
DPSRemoveFD() TableRule.eps ¬	[*nsrunLoop* removePosixFileDescriptor:forMode:]

Old Code

```
DPSAddFD(fromChild, (DPSFDProc)fdHandler,
     (id)self, NX_BASETHRESHOLD);
```

New Code

```
NSPosixFileDescriptor *filed = [[NSPosixFileDescriptor alloc]
     initWithFileDescriptor:fromChild];
[filed monitorActivity:NSPosixWritableActivity];
[filed setDelegate:self];
[[NSRunLoop currentRunLoop] addPosixFileDescriptor:filed
     forMode:NSDefaultRunLoopMode];
/* Implementation of fdHandler function from old code becomes
   implementation of the delegate method
   posixFileDescriptor:currentActivity: */
```

Miscellaneous Functions to Convert

You must also convert the functions listed in the table below by hand, as shown.

Obsolete Function TableHeadRule.eps ¬	Possible Replacement
DPSAsynchronousWaitContext TableRule.eps ¬	[*nsdpsContext* notifyObjectWhenFinishedExecuting:]
DPSPrintError(error) TableRule.eps ¬	[NSString stringForDPSError:error]
DPSPrintErrorToStream(error) TableRule.eps ¬	[NSString stringForDPSError:error]

Image View Conversion

Optional

The NeXT Application Kit provides a new class called NSImageView, which displays an NSImage. If you have an application based on the Enterprise Objects Framework that uses an NXImageView, run the **ImageViewConversion.tops** script to convert it to an NSImageView object.

There are two significant differences between NXImageView and NSImageView:

SquareBullet.eps ¬ Instead of the **style** and **setStyle:** methods, NSImageView defines **imageFrameStyle** and **setImageFrameStyle:**. This difference is significant only in that the conversion script will change all occurrences of **style** and **setStyle:** in your code to the new NSImageView methods. NSFont and NSWindow define **style** and **setStyle:** methods. You should examine your code after this conversion and make sure that the script did not change a method name incorrectly.

SquareBullet.eps ¬NSImageView adds the ability to set the alignment of the image within the view and the ability to scale the image. See the NSImageView class specification in the *Application Kit Reference* for more information on these additions.

Table View Conversion

Optional

The NeXT Application Kit provides a new class called NSTableView, which displays data in a table. If you have an application based on the Enterprise Objects Framework or the DBKit that uses an NXTableView or a DBTableView, run the **TableViewConversion.tops** script to convert it to an NSTableView object. Because there have been significant architectural changes between NXTableView and NSTableView, the usefulness of this script varies.

Changes in Architecture

NXTableView inherited from ScrollView and was made up of these components: an NXTableVector for each statically added row or column, a data source (conforming to the NXTableDataSources informal protocol) if rows or columns were dynamically added, and an NXFormatter object that drew each field along a static axis.

NSTableViews are constructed a bit differently. They inherit from NSControl, but are typically displayed in an NSScrollView. An NSTableHeaderView displays the column headings. (Rows never have headings.) Rows are always dynamically added, and columns are always statically added. An NSTableColumn object represents each column in the table view (NSTableView keeps an array of NSTableColumns). Also, each field in the NSTableView is an NSCell. Similar to NXTableView, one NSCell controls how each field in a column is drawn. The NSCell may have an NSFormatter object associated with it that specifies how the NSCell is to appear. You can use the **setDataCell:** method of NSTableColumn to set the NSCell object for that column and the **setFormatter:** method of NSCell to set the NSFormatter for the NSCell. NXFormatter is obsolete.

Selection Modes

NXTableView had a **setMode:** method that allowed you to set the selection mode: whether the user could select any of the rows or columns and whether they could select more than one at a time. With NSTableViews, users are always allowed to select rows, and you can use various methods to control whether empty selection is allowed, whether multiple selection is allowed, and whether the user can select columns too.

The **setMode:** and **mode** methods are obsolete. Because of the ubiquitousness of **setMode:** and **mode** methods in the Application Kit, the conversion script is not able to change these for you. You must look through your code to see if you send mode or **setMode:** or **mode** to an NXTableView, and if so, change it as summarized in the following table.

Obsolete Method	**Possible Replacement**
TableHeadRule.eps ¬	
[*tableView* mode] == NX_LISTMODE TableRule.eps ¬	[*tableView* allowsMutlipleSelection] == YES
[*tableView* mode] == NX_RADIOMODE TableRule.eps ¬	[*tableView* allowsMultipleSelection] == NO
[*tableView* mode] == NX_NOSELECT TableRule.eps ¬	[*tableView* allowsEmptySelection] == YES
[*tableView* setMode:NX_LISTMODE] TableRule.eps ¬	[*tableView* setAllowsMultipleSelection:YES]
[*tableView* setMode:NX_RADIOMODE] TableRule.eps ¬	[*tableView* setAllowsMultipleSelection:NO]
[*tableView* setMode:NX_NOSELECT] TableRule.eps ¬	[*tableView* setAllowsEmptySelection:YES]

Getting and Setting Values

You access data in the NSTableView by sending methods to the object that implements the NSTableDataSource informal protocol. Because columns are always static and rows are always dynamic, the information is retrieved by sending an NSTableColumn object representing the table column and an index for the row.

Obsolete Method TableHeadRule.eps ¬	Possible Replacement
getValue::into: TableRule.eps ¬	tableView:objectValueForTableColumn:row:
setValueFor::from: TableRule.eps ¬	tableView:setObjectValue:forTableColumn:

Changes to Formatting

Because NXFormatters are obsolete, the following methods are obsolete. As explained previously, each NSTableColumn has an NSCell object that controls the look of the fields in that column. Unlike the NXFormatter, the NSCell does not set any values for the field; the NSTableDataSource object does that instead.

Obsolete Method TableHeadRule.eps ¬	Possible Replacement
beginBatching TableRule.eps ¬	None
drawFieldAt::inside:inView: withAttributes::usePositions: TableRule.eps ¬	[[[[*tableView* tableColumns] objectAtIndex:*j*] €€€€€dataCell] displayRect:]

editFieldAt::	editColumn:row:withEvent:select:
endBatching	None
endEditing	[[*tableView* window] makeFirstResponder:*tableView*]
formatterAt:*i*:*j*	[[[*tableView* tableColumns] objectAtIndex:*j*] dataCell]
formatterDidChangeValueFor:at:to:sender:	None
formatterDidChangeValueFor::to:sender:	None
formatterWillChangeValueFor::sender:	tableView:shouldEditTableColumn:row: to allow editing control:isValidObject: to allow a new object
formatterWillChangeValueFor:at:sender:	tableView:shouldEditTableColumn:row: to allow editing control:isValidObject: to allow a new object
formatterWillChangeValueFor:at:to:sender:	NSFormatter, control:isValidObject:
formatterWillChangeValueFor::to:sender:	NSFormatter, control:isValidObject:
formatterDidEndEditing:endChar:	control:textDidEndEditing:
getValueAt::withAttributes::usePositions::	tableView:objectValueForTableColumn:row:
resetBatching	None

TableRule.eps ¬
TableRule.eps ¬
TableRule.eps ¬
TableRule.eps ¬
TableRule.eps ¬
TableRule.eps ¬
TableRule.eps ¬
TableRule.eps ¬
TableRule.eps ¬
TableRule.eps ¬
TableRule.eps ¬
TableRule.eps ¬

Changes to Rows

In NXTableView, if you added rows statically, an NXTableVector represented each row. NSTableViews only supported dynamically added rows. The data source controls the number of rows and which values are displayed where. The NSTableView keeps track of the current number of rows and can return the index of the selected row or the row inside a particular rectangle or containing a particular point. However, there's never an object representing the row. You can access each field in the row by sending **tableView:objectValueForTableColumn:row:** once for each column, but other than that, you never work on rows as an entity.

Obsolete Method	**Possible Replacement**
TableHeadRule.eps ¬	
addRow:at: TableRule.eps ¬	Use the data source to add rows and send reloadData.
addRow:withFormatter:andTitle:at: TableRule.eps ¬	Use the data source to add rows and send reloadData.
addRow:withTitle: TableRule.eps ¬	Use the data source to add rows and send reloadData.
dynamicRows TableRule.eps ¬	None necessary.
isRowHeadingVisible TableRule.eps ¬	None
moveRowFrom:to: TableRule.eps ¬	Use the data source to move rows and send setNeedsDisplay.
rowAt: TableRule.eps ¬	None
rowList TableRule.eps ¬	None
removeRowAt: TableRule.eps ¬	None
rowsChangedFrom:to: TableRule.eps ¬	[*tableView* setNeedsDisplayInRect:[*tableView* rectOfRow:*row*]]

rowHeading TableRule.eps ¬	None
selectRowAfter: TableRule.eps ¬	[[*tableView* selectedRowEnumerator] nextObject]
sendAction:to:forSelectedRows: TableRule.eps ¬	None
setRowHeading: TableRule.eps ¬	None
setRowHeadingVisible: TableRule.eps ¬	None
setRowSelectionOn::to: TableRule.eps ¬	selectRow:byExtendingSelection:
tableView:movedRowFrom:to: TableRule.eps ¬	None

Changes to Columns

NSTableView supports only statically added columns. An NSTableColumn represents each table. Each NSTableColumn has an NSCell object that controls how each field in the column is displayed. The NSTableHeaderView displays the column's header.

Obsolete Method **TableHeadRule.eps ¬**	**Possible Replacement**
addColumn:... TableRule.eps ¬	addTableColumn:
columnAt: TableRule.eps ¬	[[*tableView* tableColumns] objectAtIndex:]
columnCount TableRule.eps ¬	[[*tableView* tableColumns] count]
columnsChangedFrom:to: TableRule.eps ¬	[*tableView* setNeedsDisplayInRect:[*tableView* rectOfColumn:*column*]]

dynamicColumns TableRule.eps ¬	None
getValue::into: TableRule.eps ¬	tableView:objectValueForTableColumn:row:
isColumnHeadingVisible TableRule.eps ¬	headerView != nil
removeColumnAt: TableRule.eps ¬	removeTableColumn:[[*tableView* tableColumns] objectAtIndex:]
selectedColumnAfter: TableRule.eps ¬	[[*tableView* selectedColumnEnumerator] nextObject]
sendAction:to:forSelectedColumns: TableRule.eps ¬	None
setColumnHeadingVisible: TableRule.eps ¬	setHeaderView:
setColumnSelectionOn::to: TableRule.eps ¬	selectColumn:byExtendingSelection:
setValueFor::from: TableRule.eps ¬	tableView:setObjectValue:forTableColumn:

VM Conversion

Optional

Run **VMConversion.tops** if you use Mach virtual memory functions that you would like to remove from your code. The Foundation Framework provides several memory functions that you can use in place of **vm_allocate** and **vm_copy** if you sent these functions to the current task.

Obsolete Function **TableHeadRule.eps ¬**	**Possible Replacement**
round_page(*size*)	NSRoundUpToMultipleOfPageSize(*size*)

TableRule.eps ¬

trunc_page(*size*)	NSRoundDownToMultipleOfPageSize(*size*)

TableRule.eps ¬

vm_allocate(*task*, *address*, *size*, *bool*)	*address* = NSAllocateMemoryPages(*size*)

TableRule.eps ¬

vm_copy(*task*, *src*, *size*, *dest*)	NSCopyMemoryPages(*src*, *dest*, *size*)

TableRule.eps ¬

vm_deallocate(*task*, *address*, *size*)	NSDeallocateMemoryPages(*address*, *size*)

TableRule.eps ¬

vm_page_size	NSPageSize()

TableRule.eps ¬