Introduction to the OpenStep Conversion Process

This guide describes what happens when you use the automated conversion process to convert your code from NEXTSTEP° Release 3 to OPENSTEP™ Release 4.0. It also provides tips on solving some of the trickier problems that might arise as a result of the conversion.

Release 4.0 is NeXT's first OpenStep-compliant release. OpenStep is an API that enables platform-independent development of client/server applications. The OpenStep application programming interface (API) includes the Application Kit™, the DPSClient library, and a new kit called the Foundation Framework™, which provides an operating-system independence layer. The OpenStep Application Kit is functionally equivalent to the NEXTSTEP Release 3 Application Kit, but its API has been reworked to make use of the Foundation Framework.

Converting your code to this release is a good way to make your application an OpenStep application, meaning that it will run on any OpenStep system. There are a few operating-system-specific operations outside the specification's scope that your application may have to perform (such as using UNIX° sockets). However, running this conversion process will make your code as portable as possible.

This guide frequently uses the term ªOpenStepº to mean both NeXT's current release and any OpenStep implementation. NeXT's implementation of the Application Kit and the Foundation Framework provide additional features not found in OpenStep. Where these features are mentioned in this guide, they are clearly documented as NeXT features, not OpenStep features.

Major Differences between NEXTSTEP and OpenStep

The OpenStep specification replaces the NEXTSTEP 3.X Application Kit, the Common classes and functions, and the DPS Client library. OpenStep has many improvements over NEXTSTEP 3.X that are based on the design goals described in the following sections. The changes mentioned here are described further in the chapters that follow.

Portability

OpenStep contains many new classes that provide a layer of operating-system and hardware independence. These new classes provide access to operating-system services such as threads, timers, interprocess and network communication, and the current date and time. Another new class, NSString, insulates your code from the different character encodings used in different environments. All of the classes that provide operating-system independence are defined in the Foundation Framework.

Improved Support for Distributed Objects

OpenStep includes the classes that distribute objects across processes and across machines. Distributing objects improves portability because it allows you to share information between processes without using operating system calls. OpenStep also provides a protocol that allows you to use the same method to encode objects for distribution that you use to encode objects for archiving.

To allow you to distribute data more easily, many of the structures provided in NEXTSTEP Release 3 are now objects. For example, events, exceptions, colors, screens, and DPSContexts are now objects. Turning these structures into objects has other advantages: It allows you to use Objective-C° message syntax to allocate, deallocate, archive, and access this data. It also provides better encapsulation and future extensibility.

Support for Internationalized Applications

The Foundation Framework provides a new object called NSString. NSString allows you to perform character manipulation on strings without requiring that you know which character encoding is being used. Using NSStrings, you can write truly internationalized code, code that will work with any writing system supported by the Unicode standard. The Text object and many other Application Kit objects have been modified to use NSString objects in place of C strings. OpenStep also supports complete localization of an application's user interface, the messages displayed to the user, and the presentation of dates and times.

Improved Memory Management

To protect you from accessing freed objects and from never freeing your objects, OpenStep introduces a new scheme for automatically deallocating objects when they are no longer needed. This scheme uses reference counting to ensure that an object is not deallocated while it is still being used. When you create an object, you can mark it for later release. The object is then added to a pool of objects whose reference counts are decremented at the top of the event loop. When an object's reference count reaches 0, it is deallocated.

In this new scheme, an object is always deallocated by the object that created it, meaning that you don't have to worry about deallocating an object that you receive from an Application Kit object. Because the distributed objects system is now part of OpenStep, it uses this new memory management scheme and follows this same rule. Thus, the OpenStep API works the same remotely as it does locally.

Cleaner, Simpler API

Even in classes that have no conceptual changes, some improvements have been made to the API. To begin with, arguments and return values are statically typed to allow better compile-time type checking. Also, methods that used to return **self** by convention now return **void**. All classes, functions, types, and constants have the prefix ªNSº to better distinguish them from your classes and functions. Method names have been expanded so that they are more easily understood.

In addition to these API changes, which apply to virtually all classes, some of the classes with more complex

APIs have been simplified. For example, in NEXTSTEP Release 3 there are several mechanisms that allow you to control when the display of a View or Window is updated. In OpenStep, these have been condensed into a single mechanism so that many of these methods are no longer necessary.

Conversion Strategies

You convert all projects in basically the same way; however, experience level, project type, and portability issues will affect your conversion strategy. This section discusses these issues.

Converting in Stages

Note: For more complete instructions on how to convert your code, see **/NextLibrary/Documentation/NextDev/ReleaseNotes/ConvertingYourCode.rtf** on-line.

The conversion process is designed to be performed in stages. Each stage converts part of your code to the OpenStep API. There are intermediate frameworks containing header files that correspond to each stage. After you run a stage, you set the Framework Search Order to the framework for that stage, compile, and fix the errors and warnings you receive.The following figure shows what happens at each stage of the conversion.

ConversionProcess.eps ¬

When converting code for the first time, you should perform the conversion in stages. For each stage, the recommended procedure is:

1. Back up your code.
2. Start a conversion stage.
3. After the conversion is complete, use FileMerge to compare your code against the backup to learn about

the changes.

4. In Project Builder, change the Framework Search Order to the header directory for that stage.
5. Compile and fix any errors that result.

When you become more familiar with OpenStep and the issues that arise during the conversion, you may choose to run all conversions at once.

CAUTION: Even if you don't convert in stages, it is very important to follow the instructions in *Converting Your Code to OpenStep* for generating the conversion scripts. If you don't follow the instructions, you will run into trouble in the later stages of conversion.

Each conversion stage takes several minutes to complete. (Stage 1 is the longest stage.) While a stage is running, you might want to read the sections in this guide that describe the conversions being performed. The following table shows you where to locate this information. When you're fixing compiler errors, be sure to note the Gotchas sections in this guide. They provide help for the particularly tricky or obscure errors that might result from each conversion.

Stage	**Conversions Performed**	**Chapter**
TableHeadRule.eps ¬		
1	Factory Method Conversion	1
	NSName Conversion	1
	NSObject Conversion	1
	Foundation Conversion	2
	Rect Conversion	2
	String Conversion	2
	DO Conversion	4
TableRule.eps ¬		
2	Archiver Conversion	2
	Stream Conversion	2

	Icon Conversion	3
	Image Conversion	3
	Spell Checker Conversion	3
	Text Conversion	3
TableRule.eps ¬		
3	Color Conversion	3
	Event Conversion	3
	Font Conversion	3
	DPS Conversion	4
TableRule.eps ¬		
4	Printing Conversion	3
	Screen Conversion	3
	View Conversion	3
	Window Conversion	3
TableRule.eps ¬		
5	Defaults Conversion	2
	Notification Conversion	2
	Application Conversion	3
	Browser Conversion	3
	Matrix and Cell Conversion	3
	PopUp Conversion	3
TableRule.eps ¬		
6	General Naming Conversion	1
	Ivar Conversion	1
	Static Typing Conversion	1
	Void Conversion	1
TableRule.eps ¬		
Optional	Hash and StringTable Conversion	2
	List to MutableArray Conversion	2
	Stream to Mutable Data Conversion	2
	Stream to String Conversion	2
	String Conversion 2	2
	Custom IB API Conversion	4
	Image View Conversion	4

TableRule.eps ¬

Issues for Different Types of Projects

There are some special tasks for the different types of projects (application, tool, library, bundle, or palette), although the basic process is the same no matter what type of project you are converting. This section describes those special tasks.

Applications: Converting Nib Files

For applications, you must convert the nib files as well as your source code. Convert the nib files after you have converted your code but before you start to debug. For instructions on how to do so, see the on-line release note **/NextLibrary/Documentation/NextDev/ReleaseNotes/ConvertingYourCode.rtf**.

After you convert your nib files, you might want to make some adjustments to your pop-up lists. See ªPopUp Conversionº in the chapter ªConverting Application Kit Classesº for more information.

Libraries: Changing Project Types

If you are converting a library, you might want to take this opportunity to turn it into a framework. Framework is a new project type in Release 4.0 that conveniently packages a library's executable, header files, documentation, and necessary resources into a single directory. Kits are now distributed as frameworks. To convert a library to a framework, create a new project with the Framework type and copy all of the library project's files into that new project's directory.

When converting a library to a framework, you need to decide which header files you want to be public and which should be private. If a header is public, it is copied into the resulting framework directory. If the header is private, it is not copied to the frameworks directory. Headers are public by default. To declare a header private, use the File Attributes Inspector in Project Builder. Choose Inspector from the Tools menu, then choose File Attributes from the inspector pop-up.

Palettes: Converting Interface Builder API

If you have a palette project, you need to convert it before you can open the palette in the Release 4.0 Interface Builder™application. Add the framework **/NextLibrary/Frameworks/InterfaceBuilder.framework** to the palette project. This framework defines the Interface Builder API. Convert the Application Kit API by running the six-stage conversion process as you would for any other project. At the end of the six-stages, run the conversion script **CustomIBAPI.tops** to convert the Interface Builder API. The section ªCustom IB API Conversionº in the chapter ªConverting Other Kitsº describes **CustomIBAPI.tops** and summarizes the changes to Interface Builder API.

When You Have Several Related Projects to Convert

You may have several projects that need to be converted. It is important to find the correct starting point so that the conversion will go as smoothly as possible.

SquareBullet.eps ¬Convert libraries first. If you have several libraries that link against each other, convert the ªrootº library first (the one that does not link against the other libraries). Convert a library in stages and save the header files after each stage to a directory named, for example, **IntermediateHdrs***x*. Once the library is converted, convert the other projects that depend on it one by one in stages, changing the search path for header files to the appropriate **IntermediateHdrs***x* directory each time. (If you take this opportunity to change the library into a framework, you should change the Framework Search Order instead of the Header Search Order.)

SquareBullet.eps ¬Convert palettes before you convert applications that use objects defined in that palette. If you can't convert the palette and want to start with the application, you must remove all custom objects from the application's nib file before you can convert.

WARNING: The nib file for an application that uses an object from a Release 3.3 palette won't load in Release 4.0 Interface Builder. Open it in the Release 3.3 Interface Builder, remove the objects in question, and save the nib file before you start to convert.

Deep vs. Shallow Conversions

As you convert your code, you will come across places where you need to make a decision on how much you want to convert. A *deep conversion* uses as much of the OpenStep API as possible and, for this reason, produces more portable code. A *shallow conversion* preserves as much of your code as is still supported and, for this reason, takes less time than a deep conversion. For example, if you are performing a deep conversion, you run the optional conversion scripts after the six stage conversion process is complete to convert your use of Common classes to the Foundation classes. If you are performing a shallow conversion, you don't run the optional conversion scripts, as everything that the optional conversion scripts replace is still supported.

You should decide whether to do a deep or a shallow conversion based on how important it is for your code to be portable and on whether your code is going to be localized. If you want the application to run on more than one OpenStep platform, you should perform a deep conversion. If you are converting an application that is going to be localized, you also should perform a deep conversion so that localizing the application will not require changes to the code.

Note: The header file **AppKit.h** does not import all of the header files it used to. If you're performing a shallow conversion, you'll have to explicitly import header files such as **List.h**, **Storage.h**, and **HashTable.h**.

Writing Your Own Conversion Scripts

The conversion process runs a series of scripts on your code. The scripts use **tops**, a tool that performs in-place substitutions on source files according to a set of rules. The script files contain the rules that **tops** applies to your code. If it helps, you may write your own **tops** scripts to convert your custom code. To learn how to do this, read the **tops(1)** man page.

Where to Find Information on OpenStep Classes

This guide describes the changes from NEXTSTEP to OpenStep by kit. Chapter 1 describes global changes that affect all kits. Chapter 2 describes changes to the Common classes, chapter 3 the Application Kit, and Chapter 4 describes changes to other kits. (Not all kits have a conversion script.) The final chapter provides some tips for debugging your project once it's converted.

The guide focuses on changes that require you to perform some of the conversion by hand and on major conceptual changes. Trivial name changes are not documented. If you need to find out an existing method's new name, you should look at its class description in the *Application Kit Reference* or the *Foundation Framework Reference*. If the information there is not sufficient, see the *OpenStep Specification*, which you can download from NeXT's website (**http://www.next.com**). The specification provides an overview of each class. For each method in a class, it provides the syntax and a brief description.