Converting Your Code to OPENSTEP

This document describes how to convert your code from NEXTSTEP Release 3.*x* to OPENSTEP for Mach Release 4.*x*.

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 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. Because Release 4.0 is OpenStep compliant, converting your code to it is a good way to make your application an OpenStep application.

To convert your code, you run a series of scripts. These 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. Most of the

scripts are provided in the release, but you must generate some of them before you start converting because they work directly on the custom classes in your code. OPENSTEP for Mach provides a tool that allows you to generate these scripts.

The scripts convert most of the NEXTSTEP API to the new OpenStep API. Some methods, classes, and functions have been altered in such a way that an automated conversion is not possible. For these, the conversion scripts produce an error message that identifies the obsolete code and tells you how to convert it.

You run the conversion in six stages. Each stage runs a different set of scripts. After each stage, you compile your code to identify places where the conversion was not automatic. It is recommended that you run some additional scripts to replace the Common classes with OpenStep API, making your code even more portable.

This document tells you how to set up your project for conversion and how to run the conversion scripts. A separate document, the *OpenStep Conversion Guide,* describes the changes made during each of the conversion stages and the reason for those changes. Its location on-line is **/NextLibrary/Documentation/NextDev/Conversion/ConversionGuide**. Read the first chapter of this guide (**00_Intro.rtfd**) for an overview of the differences between NEXTSTEP 3.X and OpenStep and for a discussion of the different strategies you might use when converting.

The Conversion Process

To convert your code, do the following:

1. Convert your project to a 4.0 project.
2. Generate the conversion scripts.
3. Run the six-stage conversion process and any optional conversions.
4. Convert your nib files.
5. Debug your application.

These steps are described further below.

Converting Your Project to a 4.*x* Project

Before you start the conversion process, you need to change your project and makefiles so that they use the new Release 4.0 development environment. The 4.0 development environment has many significant improvements over the 3.3 development environment. In particular, Project Builder has changed significantly. To convert your project, perform the following steps:

1. Read *OPENSTEP Development: Tools and Techniques* and the release notes for Project Builder and Interface Builder to learn about changes to the environment.

   The document you are reading now does not describe how to use the new Project Builder. The book *OPENSTEP Development: Tools and Techniques* is on-line in the directory **/NextLibrary/Documentation/NextDev/TasksAndConcepts/DevGuide**. The release notes are in the directory **/NextLibrary/Documentation/NextDev/ReleaseNotes**.

2. Make sure your code compiles cleanly without warnings.

   Perform this step so that any warnings that show up during conversion won't become mixed in with pre-existing warnings. Make sure the **-Wall** compiler option is being used.

   Before you compile your 3.3 code on a 4.0 system, you need to change the search path for header files to the directory **/NextDeveloper/OpenStepConversion/3.3Headers**. To do this in the new Project Builder:

   1. Choose Inspector from the Tools menu to bring up the project inspector.

2. From the inspector pop-up list, choose Build Attributes.
3. Choose Header Search Order from the pop-up list in the Build Attributes inspector.
4. Type **/NextDeveloper/OpenStepConversion/3.3Headers** and click Add.

3. Back up your project directory.

   The conversion scripts modify your code in-place, and they do not create backups for you. It is strongly recommended that you back up your project after each conversion stage in addition to backing up before you begin, provided you have enough space.

4. Delete the NeXT-provided libraries (such as **libNext**) from your project, and add the corresponding frameworks (such as **/NextLibrary/Frameworks/AppKit.framework**).

   All NeXT-provided libraries are replaced with frameworks in 4.0. A frameworks' executable code is a dynamic shared library. To add the Application Kit framework, in Project Builder choose Add File from the Project menu, select **/NextLibrary/Frameworks/AppKit.framework** from the Add File panel, and choose Frameworks from the file type pop-up list on the Add File panel. Add the Foundation Framework in the same way.

   Frameworks are bundled differently than shared libraries. All of the support files for a framework are contained in the same directory, so you no longer

have to know that the library itself resides in one place, its header files another place, and its documentation still a third place.

When you add a framework to your project, it doesn't appear in the Libraries suitcase. Instead, it appears in the Frameworks suitcase. You are able to see that framework's header files and documentation underneath that suitcase.

5. Change the application class in Project Builder.

   In OpenStep, all keywords are prefixed with "NS", so the Application class is now NSApplication. If you are converting an application project, you need to change the class in Project Builder so that it will use the appropriate class when updating your **main** function. To do this, bring up the project inspector by choosing Inspector from the Tools menu. In the inspector, choose Project Attributes. Finally, type NSApplication in the field labelled Application Class.

6. Convert your makefiles.

   The new Project Builder comes with new **Makefile.postamble** and **Makefile.preamble** files. Convert your preamble and postamble files if you performed some customization on them. You can find copies of the new templates in the directory **/NextDeveloper/Makefiles/project**. Rename your existing preamble and postamble files, copy the new templates into your project directory, rename the template (remove the **.template** extension), and merge any customizations you would like to keep into the new templates.

Most of the changes to the makefiles are additions made to support frameworks, however some makefile variables are now obsolete. The additions are documented in the comments in the preamble and postamble files. The following table lists the obsolete makefile variables and what you should use as a replacement.

Obsolete Makefile Variable	Possible Replacement
BUNDLELDFLAGS	OTHER_LDFLAGS
PALLETTELDFLAGS	OTHER_LDFLAGS
COMMON_CFLAGS	OPTIMIZATION_CFLAG, WARNING_CFLAGS
NORMAL_CFLAGS	OPTIMIZATION_CFLAG, WARNING_CFLAGS
DEBUG_CFLAGS	Use DEBUG_BUILD_CFLAGS.
PROFILE_CFLAGS	Use PROFILE_BUILD_CFLAGS.
DYLD_APP_STRIP_OPTS	LIBRARY_STRIP_OPTS (-S by default)
RELOCATABLE_STRIP_OPTS	DYNAMIC_STRIP_OPTS (-S by default)
OTHER_DEBUG_LIBS	Add libraries using Project Builder.

OTHER_PROFILE_LIBS	Add libraries using Project Builder.
OTHER_JAPANESE_DEBUG_LIB	Add libraries using Project Builder.
OTHER_JAPANESE_PROFILE_LIBS	Add libraries using Project Builder.
BUNDLE_LIBS	Add libraries using Project Builder.
PRECOMPS	Use Project Builder Inspector to mark headers for precompiling.

In addition, the following are some other changes to makefiles of which you should be aware:

- If your project is a library shared by several other projects, consider converting it to a framework. For more information, see the book *OPENSTEP Development: Tools and Techniques*.
- Many things that you used to have to set using the Makefile preamble and postamble files you can now set using Project Builder. For example, you can set search paths and simple compiler flags. You should minimize your use of the preamble and postamble files and use the Project Builder interface instead. For more information, see the development environment release notes.
- The build process now uses **gnumake** instead of **make**.
- If you wrote any top-level double-colon targets that are also implemented by Project Builder, such as **app::**, **install::**, **all::**, now is a good time to rename

them. It is likely that the variables passed to these rules and the order in which they are executed has changed. Consider using **after_install::**, **before_install::**, **OTHER_INITIAL_TARGETS**, or **OTHER_PRODUCT_DEPENDS**.

- Once you save your project in 4.0, it uses the new makefile behavior. If your project must use the 3.3 makefile behavior, set **MAKEFILEDIR** in **Makefile.preamble** to **/NextDeveloper/Makefiles/app**.

Generating the Conversion Scripts

Some of the conversion scripts need to understand your application's class hierarchy and how you implemented certain methods. For this reason, you must generate some of the conversion scripts yourself.

To generate conversion scripts, do the following:

1. Add **/NextDeveloper/OpenStepConversion/UtilityScripts/shellscripts** to your **PATH** environment variable. The **convert** command, which you use to convert your code, is in this directory.

2. In a Terminal window, enter these commands to create files that will be used to generate the conversion scripts:

```
% cd project_directory
% convert -preprocess
```

3. Modify the files **StringMethods**, **StringDefines**, **RectMethods**, and **VoidMethods**, which are used to generate the conversion scripts. Carefully read "Modifying the Files Used to Generate Conversion Scripts" below for instructions on how to do this. Once you perform step 4, you cannot return to this step.

WARNING: Do not move on to step 4 until you have fully completed step 3.

4. Generate the optional conversion scripts with this command:

```
% convert -makescripts
```

Modifying the Files Used to Generate Conversion Scripts

The **convert -preprocess** command creates a **CONVERSION** directory under your project directory and stores in it files that describe your application. These files and their contents are described in the table below.

File	Description

ClassHierarchy1	Describes the class hierarchy before conversion in a format **tops** can understand.
ClassHierarchy2	Describes the class hierarchy after conversion in a format **tops** can understand.
StringMethods	Lists methods that have **(char *)** or **(const char *)** as either a parameter type or return type.
StringDefines	Lists **#define**s that were determined to be of type **(const char *)**.
RectMethods	Lists methods that use pointers to NXRects, NXSizes, or NXPoints as either a parameter or a return type.
VoidMethods	Lists methods that did not specify a return type.

In order to generate conversion scripts that will convert your code properly, you need to modify all of the files produced by **convert -preprocess** except the class hierarchy files. The following sections provide instructions on this step.

After you modify these files and run **convert -makescripts**, some additional conversion scripts appear in the **CONVERSION** directory. These scripts are executed during the conversion process.

StringMethods

OpenStep 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 portable and internationalized code, code that will work with any writing system supported by the Unicode standard. The Application Kit now uses NSStrings where it used to use C strings in method and function arguments and return values. Because of this and because of the advantages of NSStrings, you may want to convert all of the C strings in your application so that they use NSStrings.

The **StringMethods** file contains a list of every method in your application that takes a C string as an argument or returns a C string. It does not list overrides of Application Kit methods; those are taken care of by the conversion scripts provided in the release. All of the methods listed in this file will have their C string arguments and return values converted to NSStrings.

Only methods that don't modify the string should have their C strings converted. Look at the implementation of every method listed in the **StringMethods** file. If the method modifies its C strings, remove its name from the file. To skip the optional conversion of C strings to NSStrings entirely, delete all of the methods from this file. (You may want to save them in a different file.) Before you decide, you may want to read about the conversion of C strings to NSStrings in the *OpenStep Conversion Guide*. See the chapter "Converting the Common

Classes."

You can locate a method's implementation easily in Project Builder by doing the following (for more information, see the development environment release notes):

1. Choose Find from the Tools menu to display the Project Find panel.
2. In the Project Find panel, enter the name of the method in the text field.
3. Make sure Definition is selected in the pop-up list.
4. Click the Find button.

StringDefines

This **StringDefines** file contains **#define** macros that are string constants or **NXLocalizedString...** function calls. These macros will be converted so that they create NSString objects instead of C strings. If this file lists macros that you want to remain C strings, delete the line naming that macro from the **StringDefines** file.

RectMethods

All Application Kit functions and methods that used to take the address of an NXRect or an NXSize now take the value of the structure. Similarly, all

Application Kit functions and methods that used to return a pointer to an NXRect or NXSize now return the structure itself. This change was made to eliminate the aliasing problems that can occur when you pass pointers and to allow these methods to work better with the Distributed Objects system.

The **RectMethods** file contains a list of every one of your application's methods that takes a pointer to an NXRect, NXSize, or NXPoint structure or returns a pointer to one of these structures. (It does not list overrides of Application Kit methods.) All of the methods listed in this file will have their structure pointer arguments and return values converted to the actual structure.

Only methods that don't modify the structures should have their arguments and return types modified. Look at the implementation of every method listed in the **RectMethods** file. If the method modifies an NXRect, NXPoint, or NXSize structure, remove its name from the file. To skip the optional conversion of structure pointers entirely, delete all of the methods from this file. (You may want to save them in a different file.) Before you decide, you may want to read about the NXRect, NXPoint, and NXSize conversions in the *OpenStep Conversion Guide*. See the chapter "Converting the Common Classes."

VoidMethods

Previously, methods returned **self** by convention. Some methods return **self** to

indicate success and **nil** to indicate failure. Returning **self** to indicate a Boolean value or returning **self** without any associated meaning made the API more confusing. In OpenStep, when a method has no real value to return, its return type is **void**. Where a method returned **self** or **nil**, it now returns **BOOL**. In addition to being cleaner API, returning **void** and **BOOL** helps you avoid creating unnecessary proxies if you're distributing objects. The **VoidMethods** file contains a list of every method in your application that has no return type specified (except for overrides of Application Kit methods).

Look at both the implementation and the uses of each of the methods listed in the **VoidMethods** file and perform the action listed below. (To look at all of the uses of a method, enter its name in Project Builder's Project Find panel, choose References from the pop-up list, and click Find. The bottom half of the Find panel lists all of the places the method is invoked.)

- If the method returns either **self** or **nil** and invocations of the method test the return value, change the prefix for that method in the **VoidMethods** file to **SELFNIL-BOOL**. This will convert the method to return **BOOL**. (By default, all methods in the file are prefixed with **SELF-VOID**, which means they will be converted to return **void**.)
- Delete the method from the **VoidMethods** file if its return type is ever used in any of the places where it is invoked (and it should not be converted to return **BOOL**).

- **init...** methods do not appear in the **VoidMethods** file because it is correct for them to return type **id**. If you have a method that starts with **init...** but is not a initialization method for its class, add it to the **VoidMethods** file if it should return **void**.
- Leave all **+initialize** methods in the file. The **+initialize** method now returns **void**.
- If the method returns a value other than **self** and that value is never used anywhere the method is invoked, change its prefix to **OBJ-VOID**.
- To skip the optional void conversion, delete all of the methods from the **VoidMethods** file. (You may want to save them to a different file.)

For example, consider the methods shown in the following code fragment. All of these methods except **newCount** would be listed in the **VoidMethods** file because they don't specify a return type. However, the **countingObject** method has a meaningful return value because **countSomething** expects it to return an object. **countSomething** is is the only method that should truly be converted to return **void** because it is invoked as if it already did return **void**. Thus, in the **VoidMethods** file, you would delete **countingObject** but leave **countSomething**.

```
- countingObject
{
   return countingObject;
```

```
}

- countSomething
{
   [[self countingObject] incrementCount];
   return self;
}

- (int)newCount
{
   [self countSomething];
   return [countingObject currentCount];
}
```

Before you decide which methods should be converted to return **void**, you may want to read about the **void** conversion in the *OpenStep Conversion Guide*. See the chapter titled "Global API and Style Changes."

Running the Conversions

The conversion process is organized into six stages. Each stage runs a series of scripts on your code and makes changes based on the information in those scripts. When converting code for the first time, you should perform the conversion in stages. Here's the recommended procedure:

1. Back up the project directory.

2. In Project Builder, close all of the source files.

   The `convert` script, which you run in the next step, changes your source files. If you have looked at these same source files in Project Builder before you run the script, the file displayed by Project Builder won't reflect the changes that `convert` makes. To make sure that you are always looking at the latest version of your files, close all of the files in Project Builder before you run the script.

   You can see which files you have loaded in Project Builder using the Loaded Files panel. Choose Loaded Files from the Tools menu to bring up the Loaded Files panel. Select a file in the Loaded Files panel, then choose Close from the File menu to close it.

   If you don't perform this step, `convert` will still work properly, but when you look at the source files after it is complete, they won't reflect the changes. To see the changes that `convert` made, press Command-u.

3. In a Terminal window, enter:

   ```
   % cd project_directory
   % convert -stagex
   ```

   where *x* is the number (1 through 6) of the conversion stage you want to

perform. If you don't specify source files on the command line, they will be identified with the pattern:

```
*.[hcmCM] *.psw* *.*proj/*.[hcmCM] *.*proj/*.psw*
```

In other words, all code files in your current directory and in the first level of subproject directories will be converted. If this is not sufficient, specify the appropriate file list after the **-stage** option:

`convert -stage`*x file1 file2* ...

Each conversion stage takes several minutes to complete. Stage 1 is the longest stage.

WARNING: The conversion scripts should not be run out of order, and it is not recommended that they be run on the same file more than once. In general, it's a good idea to save a copy of your files after each stage.

4. Once the conversion is complete, use FileMerge to compare your project with the backup of your project. This is a good way to learn about the differences between NEXTSTEP and OpenStep. (To learn how to use FileMerge, see the development environment release notes.)

5. Open the project in Project Builder and use the Inspector panel to add **/NextDeveloper/OpenStepConversion/IntermediateFrameworks***x* to the search path for frameworks, where *x* is the number of the conversion stage.

(For stages 2 through 6, remove the **IntermediateFrameworks** directory for the previous step.)

In Project Builder, you change the search paths for frameworks from the Build Options inspector. Choose Inspector from the Tools menu to bring up the inspector panel, and choose Build Options from the inspector's pop-up list. In the Build Options inspector, there is another pop-up list. Choose Framework Search Order from that list. Type the new search path for frameworks, then click Add.

In between the first conversion stage and the last conversion stage, your code is in an interim state and will not compile successfully with either the NEXTSTEP 3.X headers or the OPENSTEP for MachOS Release 4.0 headers. The **IntermediateFrameworks** directories have NEXTSTEP headers at the intermediate stages of conversion so that your code will compile. Your code will not link successfully until after the sixth stage.

There is no **IntermediateFrameworks** directory for stage 6. After stage 6, you should use the default search path for frameworks (**/NextLibrary/Frameworks**).

NOTE: Be sure you change the Framework Search Order, not the Header Search Order.

7. Build your project and work through any error messages.

The conversion process places **#error** messages in places where automated conversion was not possible and **#warning** messages in places where the conversion might not be correct. You will see these messages when you compile. Once your code compiles successfully, back up your project again, and run the next stage.

If you need help deciding how to correct an error, see the *OpenStep Conversion Guide*. It describes how to correct most of the errors that occur. If you need more information about a particular class or function, look in the *Foundation Framework Reference* in **/NextLibrary/Frameworks/Foundation.framework/Resources/English.lproj/Documentation** or the *Application Kit Reference* in **/NextLibrary/Frameworks/Foundation.framework/Resources/English.lproj/Documentation**. If the class has no documentation yet, see the *OpenStep Specification* in **/NextLibrary/Documentation/OpenStepSpec**.

NOTE: If Project Builder is not showing you the updated source files (the source code does not match the warning you see), type Command-u. Be sure to close all files in Project Builder as described in Step 3 before you run the next conversion stage.

You may want to set the Continue After Error preference to Project Builder. If you do, Project Builder will continue to build even after it finds an error. You can find the Continue After Error preference in the Preferences panel, under Build.

8. If you're converting an application project, convert your nib files. To do this, use this command:

```
% convert -nib
```

Running All Conversions At Once

After you have converted at least one project and you are more familiar with the conversion, you may want to run all conversions at once. To do this, enter this command in the terminal window:

```
convert -all
```

This command runs all six conversion stages one by one, then converts any nib files. Before you enter this command, you still must generate the conversion scripts for the project as described earlier in this document. Remember to be very careful when modifying the files used to generate the conversion scripts. It will save you time in the end.

Running the Optional Conversions

After you have completed all of the conversion stages you may wish to run remaining optional conversions. The optional conversions are listed below.

Conversion Script	**Purposes**
CustomIBAPI.tops	Converts Interface Builder API for palettes.
ListToMutableArray.tops	Converts List objects.
HashAndStringTableConversion.tops	Converts HashTable and NXStringTable objects.
StringConversion2.tops	Converts more C strings to NSStrings.
StreamToMutableData.tops	Converts streams to NSMutableData objects.
StreamToString.tops	Converts streams to NSString objects.
TableView.tops	Converts NXTableView objects to NSTableView objects.
VMConversion.tops	Converts MachOS virtual memory functions to Foundation functions.

To run a single script on your source code, use this command:

```
tops -scriptfile scriptFile *.[hcmCM] *.psw* *.*proj/*.[hcmCM]
*.*proj/*.psw*
```

Where *scriptFile* is the complete path for the script (**/NextDeveloper/OpenStepConversion/ConversionScripts/***scriptName*). See

the *OpenStep Conversion Guide* for information on these scripts.

NOTE: Because the API changes between C strings and NSStrings and between NXTableViews and NSTableViews are significant, the usefulness of the scripts **StringConversion2.tops** and **TableView.tops** vary. You may want to run them on a copy of your code first to see if they help you.