Porting Guide

Building Your Project on Windows

1. Copy the project files from the Mach system to the Windows system.
2. Build the project on the Windows system.
3. Run, test, and debug the application.

Once that you've ensured that the code in your OPENSTEP project is portable to Windows, you're ready to build it on your Windows NT system.

Copying the Project

OPENSTEP for Windows includes an installable package containing Samba, a file server that exports the UNIX file system to other systems, such as Windows NT. With Samba, an OPENSTEP computer can share directories and printers with computers running Microsoft Windows. Once Samba is installed and configured, you can use Window's Explorer or File Manager programs to copy your application project from your OPENSTEP for Mach system to your OPENSTEP for Windows system. The Samba package contains configuration instructions.

You can also use **rcp**, **ftp**, or a similar file-transfer tool to copy your project from your OPENSTEP for Mach system to your OPENSTEP for Windows system. Issue the command from the Windows system, preferably from the directory in which you want the project directory to be.

1. Double-click the Bourne shell icon (located in the NeXT Software program group) to start up a shell.

2. Connect to the directory where you want the project to go. Create the directory if that's necessary (by choosing, in the File Manager, Create Directory from the File menu) and set the necessary permissions.

3. Copy the project from your OPENSTEP for Mach system.

   The following **rcp** example copies the project Invoicer from host machine workhorse:

   ```
   $ rcp -rb workhorse:/Projects/Invoicer .
   ```

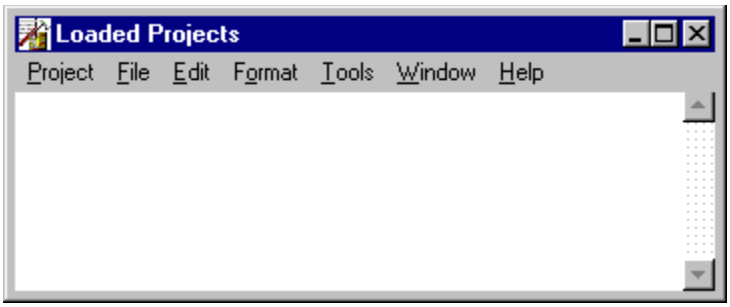
   The -b (binary) flag is necessary to prevent binary data files (such as nib files) from being corrupted by the CR/LF substitution that is performed otherwise.

Building the Project

Once you've copied the project to your OPENSTEP for Windows system, you can build it using Project Builder:

1. **Launch Project Builder**: Locate **ProjectBuilder.exe** in **$(NEXT_ROOT)/NextDeveloper/Apps/ProjectBuilder.app** and double-click it to launch it.

2. **Open the project**: From Project Builder's Loaded Projects window, choose the Open command from the Project menu:



In the Open dialog, navigate to your project directory, select the **PB.project** file, and click OK.

3. **Build the project**: Click the Build icon (the hammer) on Project Builder's main window. Click the same iconic button again on the Project Build panel. Project Builder will commence building your application.

You can also build your project from a Bourne shell:

1. Connect to the project directory.
2. Issue the **make** command against the target all, specifying the OBJROOT and SYMROOT directories (temporary directories to hold files generated by the build). Here's an example:

```
$ make all OBJROOT=c:/temp/obj SYMROOT=c:/temp/sym
```

When you issue this command, the make utility proceeds to build your project, and emits compiler and linker messages as it proceeds.

Interpreting Compiler Output

When your application compiles, you may see messages like this:

```
Controller.m: In function _i_Controller__openFile_withFlags_

Controller.m(153): warning: suggest parentheses around assignment
used as truth value
```

The compiler transforms Objective-C methods into functions with the following structure:

```
_i_Controller_DataSource_openFile_withFlags_
```

The first element indicates whether this is an instance method (i) or a class method (c). Following this indicator is the class name, the category (if any), and the method name. If there is no category, you'll just see two underscore characters. Underscores also replace the colons in method keywords.

Knowledge of this structure will aid you in debugging your code, as you'll see later.

Testing and Debugging the Application

Once your application compiles and links, run it and see how it behaves. Locate the application executable in the **.app** directory, which itself is either in the project directory or, if you issued the **make** command from the command line, in the SYMROOT directory that you specified. Double-click

the **.exe** file to launch the application. Give the application a test drive. See what works and what doesn't, if anything.

Debugging the Application

You can debug your application using NeXT's version of **gdb,** the GNU debugger. Currently, you must run **gdb** from a Bourne shell:

1. Open a shell by double-clicking **sh.exe** in **$(NEXT_ROOT)/NextLibrary/Executables**.

2. Change your working directory to your project directory, for example:

   ```
   cd /Projects/MyAppProject
   ```

3. At the command prompt, enter "gdb" followed by the application's "app" directory and the executable (which, of course, will contain more symbolic information if the build target is debug).

   ```
   gdb MyApp.app/MyApp.exe
   ```

4. Run the debugger and debug the program. See the GNU Debugger Reference for details of usage.

Note: If your executable contains CodeView symbols, you can also debug your Objective-C application using Visual C++. For details, see Using Visual C++ to Debug Objective C.

Command Line Debugging Flags

When you start up an application from the command line you can specify flags that will help you debug the application.These flags, described in the table below, make available details of behavior related to the Window Server and PostScript generation.

You can specify the debugging flags from the command line or in Visual C++. This example shows an application run from the command line with two debugging flags:

```
$ MyApp -NSSyncPS YES -NSShowPS YES
```

To specify the flags in Visual C++:

1. Choose Settings from the Project menu.
2. Click the Debug display.
3. Type the debugging flag or flags in the Program Arguments field.

Flag	**Effect**
NSSyncPS	If set to any non-zero value, makes the application wait for the Window Server. Whenever the application sends PostScript code to the Window Server, it will wait for the code to be executed before proceeding. This results in error messages being more closely associated with the code that produced them.
NSShowPS	If set to any non-zero value, causes all PostScript code sent to the Window Server to be also written to **/tmp/console.log**. The **/tmp** directory must already exist.
NSPSDebug	If set to YES, causes an alternate PostScript error-handling

	routine to be installed. This routine produces more detailed debugging information, including the contents of the operand stack.
NSShowAllWindows	If set to a non-zero value, forces all windows to be always on-screen. Windows that are normally hidden, such as windows that store images that are composited to other windows, will be visible as your program runs.
NSAllWindowsRetained	If set to a non-zero value, forces all buffered windows to be retained windows. Since drawing is done directly on an on-screen retained window, youll be able to see PostScript code being rendered. (In a buffered window, drawing is rendered in a buffer and then is flushed to the window.)

Some Common Problems

If your application fails to launch, crashes, or behaves unexpectedly when you first try to run it, you can often quickly determine the reason. Follow these steps to isolate and resolve the problem:

1. Look in the application log in the Event Viewer.

   From the Program Manager open the Administrative Tools program group and double-click the Event Viewer application. Choose Application from the Log menu and double-click the first item in the log.

2. If the problem is related to Display PostScript, make sure **WindowServer.exe** is running.

3. If the problem is one of the following, make sure that **pbs.exe** is running and that **NEXT_ROOT/NextLibrary/Fonts** and **NEXT_ROOT/NextLibrary/Rulebooks** contain the proper files.

   The application cannot connect to **pbs**.
   The application cannot get font information.
   The application takes a long time to run.
   The application has no text when it appears.

4. Make sure that the necessary DLLs are in the proper directories: **nextpdo.dll**, **Foundation.dll**, **AppKit.dll**, and other DLL's related to installed frameworks go in **NEXT_ROOT/NextLibrary/Executables**.

   From the Program Manager, check the System control panel to verify that these directories are defined for the user environment variable Path.

5. An application can fail to launch because its main nib file was created with an earlier version of Interface Builder. To resolve this problem, open your applications nib files on your OPENSTEP for Mach system, make some trivial change to mark the files as dirty, re-save the files, and copy them to the appropriate source **.lproj** directory on your OPENSTEP for Windows system.

OPENSTEP 4.0 for Windows

Porting Guide

This guide contains instructions for porting OpenStep for Mach 4.0 (NEXTSTEP 4.0) applications to OpenStep for Windows 4.0.

Contents

Porting Guide

Using Visual C++ to Debug Objective-C

In addition to using **gdb**, the GNU debugger, you can use the Microsoft Visual C++ debugger to debug your Objective-C code on NT. The **gcc** compiler on Windows, if given the **-gcodeview** flag, generates CodeView debugging information that can be used by the Microsoft Visual C++ debugger or by any other Windows debugger that supports CodeView symbols.

The Visual C++ debugger is not programmable or otherwise extensible and thus there is no way to provide the same level of debugging support that **gdb** provides. Here are some of the limitations that you will experience when you use Visual C++ to debug Objective-C code:

* You can't invoke functions from the command line.

* You can't set breakpoints by method name. However, you can set breakpoints on methods using the equivalent function name generated by the compiler.

* You can't directly inspect an Objective-C object. However, there are ways to indirectly inspect objects.

* When you view an object (whether global or local) you can only look at a snapshot of the object. You don't get its superclass information by default. You can only see the scope of the one class unless you explicitly cast it.

* If something is typed as **id**, you have to explicitly cast it to display information about it. Because of this, you may want to use static typing whenever possible.

* You can't use the debugger to debug NSThreads.

You can step into message expressions by stepping into **objc_msgSend** calls in the Disassembly panel. Although this technique (explained below) is not simple, it is effective. Otherwise, you can only single-step and set break points on a method (though once you set a break point on a method you can step through it).

To debug your Objective-C code in Visual C++:

1. Build a version of your program that includes debugging information by including **-g** on the **gcc** command line (this is the makefile default).

2. Open the resulting **.exe** file in Visual C++, either by dragging it onto the Visual C++ icon or by explicitly opening it from within Visual C++ by choosing Open from the File menu. If you have problems with this step, simply restart the debugger from within Visual C++.

3. Open your source file in Visual C++.

4. Set any breakpoints.

5. Choose Go from the Debug menu.

The display changes to include the Locals and Watch windows. The Locals window lets you examine local variables, while the Watch window lets you examine specific variables and expressions.

Inspecting Objective-C Objects

You use the text field in the Watch window to examine and change variable values. Using this field, you can inspect an Objective-C object by first casting the object to a C structure and then examining the structure's members. Similarly, by typing the appropriate information in the QuickWatch field, you can retrieve an objects value, its class, and its data members.

For example, suppose you have following simple application:

```
#import <Foundation/Foundation.h>

int main (){
    int num = 33;
    char name[256] = "Lois Lane";
    id firstString;
    NSString *secondString = @Clark Kent;
    id nsnum;

    firstString = [NSString stringWithCString:name];
    nsnum = [NSNumber numberWithInt:num];
    NSLog(@"The string is %@\n", firstString);
    NSLog(@"The number is %@\n", nsnum);
    return 0;
}
```

Typing the following sample statements in the Watch text field would have these effects:

Watch Statement	Effect
nsnum	Returns the location in memory where the object was allocated.
(NSString*)firstString	Returns the statement +(NSString*)string = <hex address>. The plus sign indicates that the statement can be expanded to display additional information. To expand a statement, double-click it.
(NSString)firstString	Has the same effect as the previous statement, except that Watch represents the objects contents as a structure ({...}). As with the above, you can expand it to display more detail about structure members.
(char*)(NSString*)firstString+8	Casts the variable firstString to a C string and displays its value, Lois Lane. This works for NSInlineCStrings, but is not guaranteed to work for other string types such as NXConstantStrings (which is what secondString is).
(char*)((struct objc class*)firstString)->isa->name	Returns an ASCII representation of the variable strings class. Note that this works for local variables only; instance variables must be accessed through self.
(char*)(NSString*)secondString->isa->name	Same as the above. This works for secondString but not for firstString because secondString was statically declared to be an NSString. Because secondString is an NXConstantString,

	you cant use a statement such as (char*)(NSString*)string+8 to display its contentsthat only works for NSInlineCStrings.
*(char **)(secondString+4)	If you have an NSCString or an NXConstantString (like secondString) you can retrieve its string value by making the casts illustrated here (after the **isa** pointer is a pointer to the raw bytes).

To display information about instance variables, you must access those variables through self. For example, if you have an instance variable length, you can display information for it in the Watch window by typing the following:

```
(char*)((struct objc_class*)self->length)->isa->name
```

Setting Breakpoints by Name

You can set breakpoints on methods by name, but not by the Objective-C name that identifies them in your source code. You must use the function name that the compiler assigns to the method. As explained in Interpreting Compiler Output, above, these function equivalents have the form:

```
_i_MyClass_MyCategory_method_name_
```

The initial "i" means instance method ("c" indicates class method). When theres no category, two underscores are between the class and the method. Underscores are substituted for the colons terminating method keywords.

To set a breakpoint on a method by name, type it with the appropriate function format into the Location field of the Breakpoints panel and click Add. The application must be compiled with the **-g** flag specified (the make default).

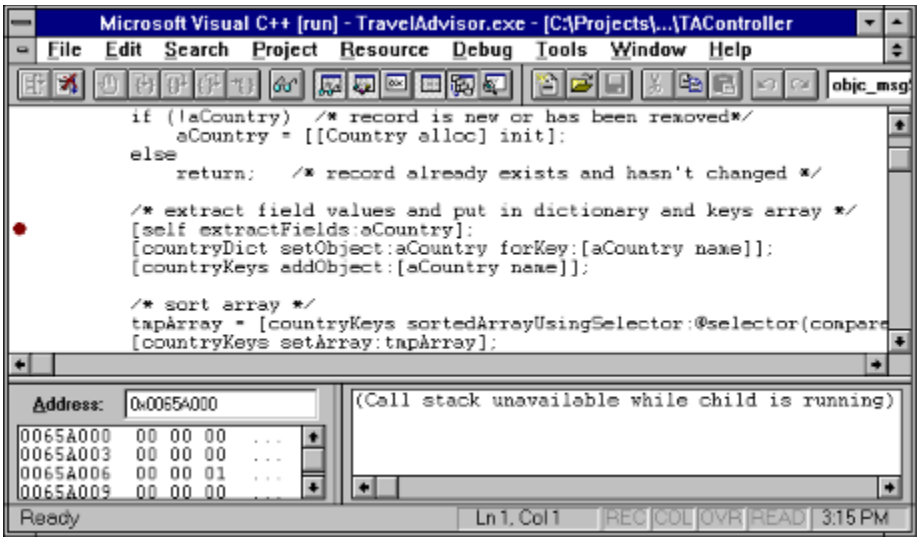
Stepping into objc_msgSend

Using the Visual C++ Disassembly panel, you can step into an invoked method via the **objc_msgSend** call and exit into the source code implementing that method. The methods susceptible to this technique can only be your own, not methods in libraries for which you dont have source code, such as the Foundation framework. To step into methods:

1. Set a breakpoint on or near the invocation of the method you want to step into.

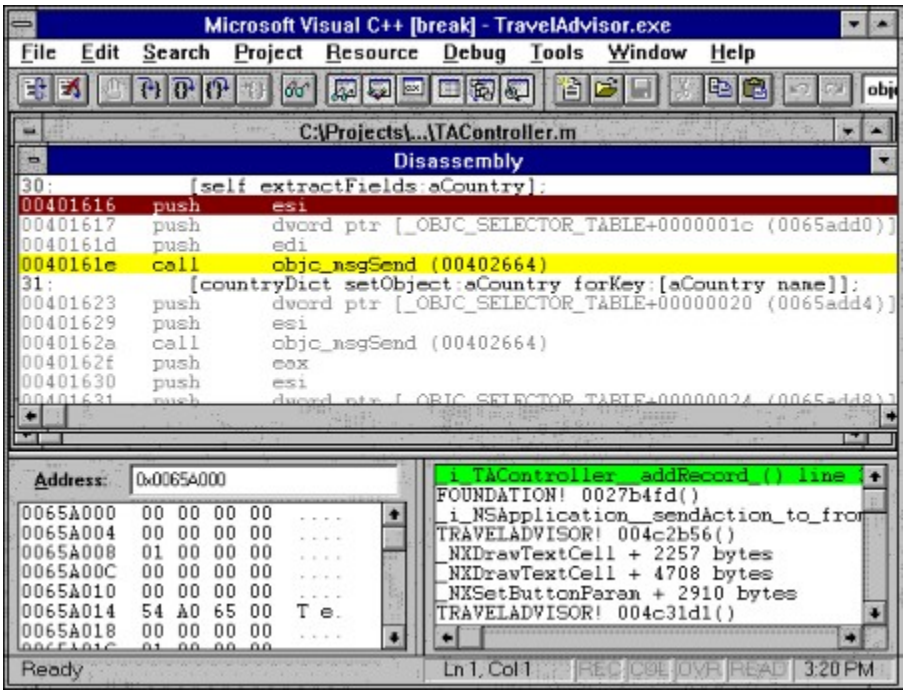
2. Run the program to that breakpoint.

3. Choose Step Over from the Debug menu (or click the associated icon) until the cursor is on the line containing the method you want to step into.



4. Choose Disassembly from the Debug menu.

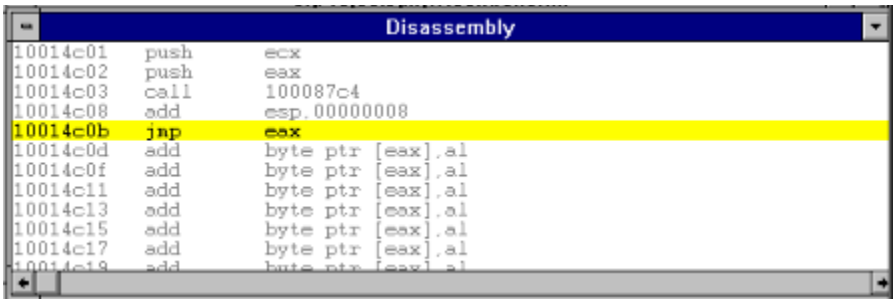
5. In the Disassembly panel, step over lines until you get to the first **objc_msgSend**.



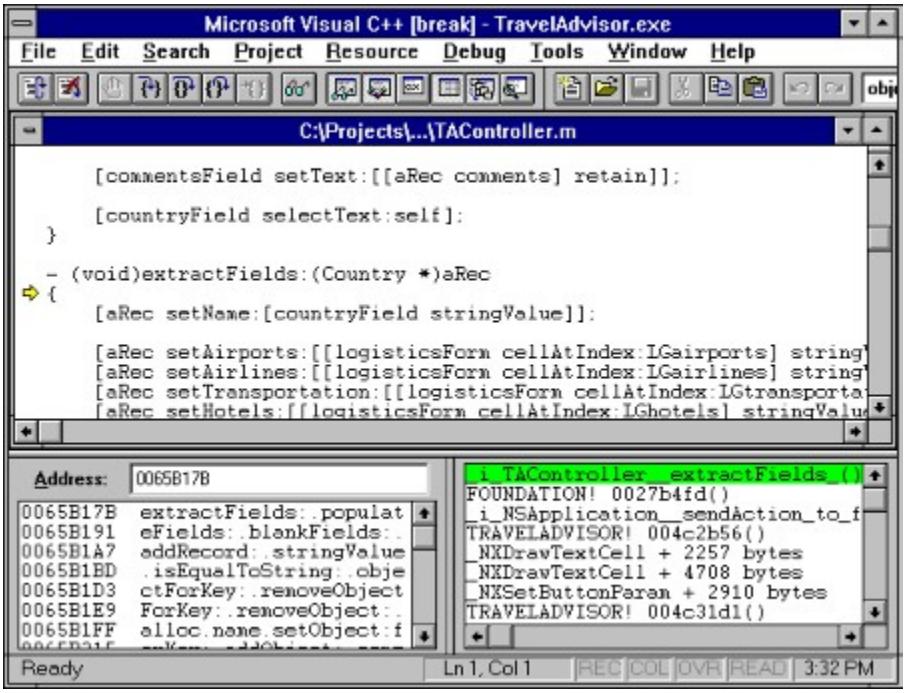
6. If you want to verify the associated selector at this point, follow the instructions in Identifying Selectors, below.

7. Choose Debug->Step Into (or click the associated icon) to step into the **objc_msgSend** call.

8 Step over lines until you come to **jmp eax**; step into it. This assembly directive actually switches the thread of execution to the invoked method.



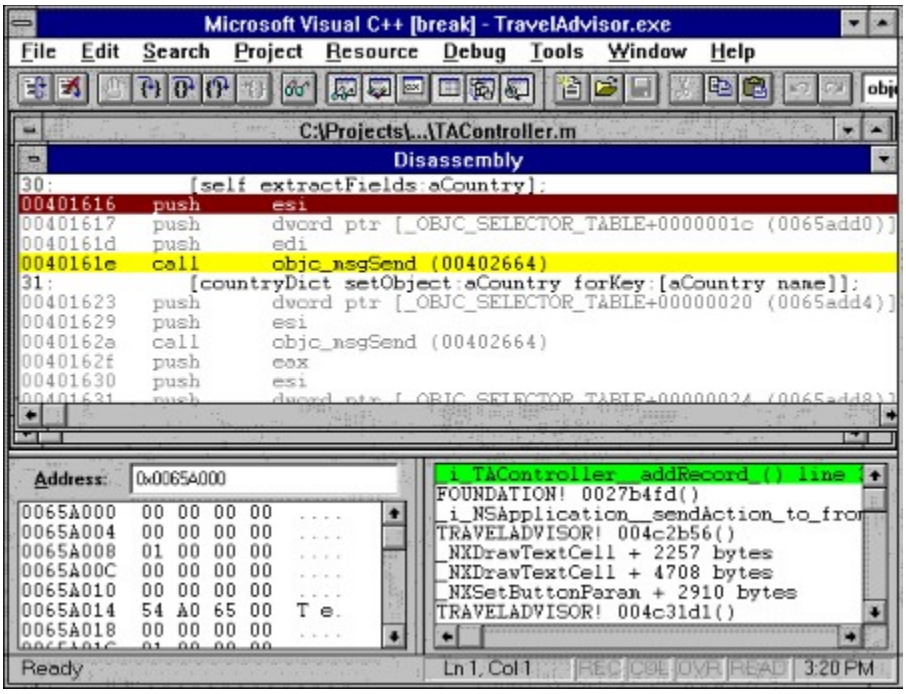
9. You're now in the portion of assembly code related to the invoked method's first line of source code. Choose Debug->Disassembly to toggle off the Disassembly panel and pop up the source code file.



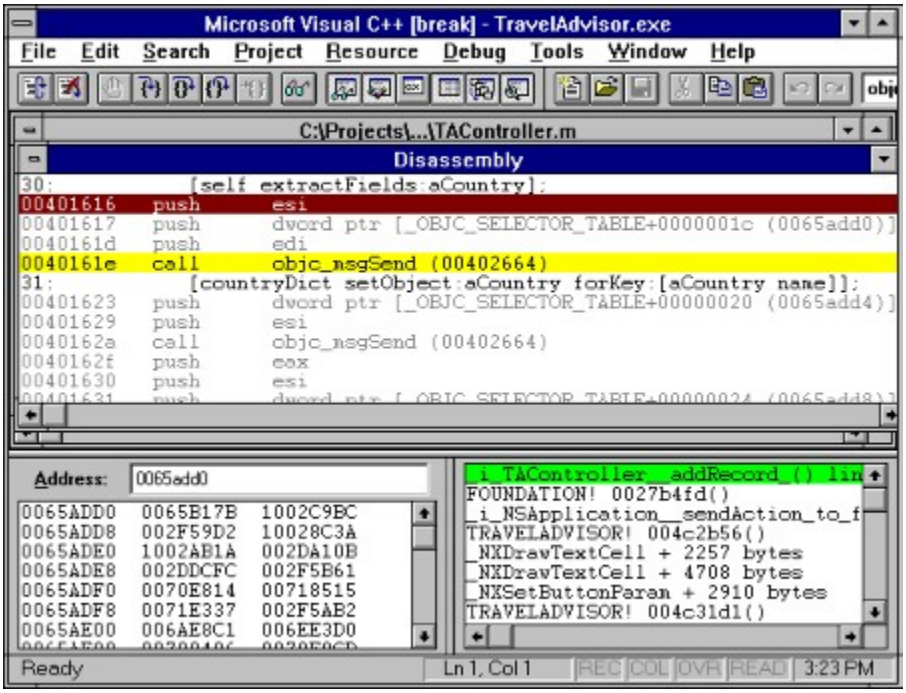
Identifying Selectors

To identify the method selector associated with an **objc_msgSend** in the Disassembly panel, do the following:

1. In the Disassembly panel, before you step into **objc_msgSend**, locate the selector stack just before it. Selector stacks hold the selector for the next **objc_msgSend** call or for a series of **objc_msgSend** calls if messages are nested.



2. Double-click the address in the parentheses to the right of the selector stack to select it, then drop it to the Memory panel



3. Since the address you drop onto the Memory panel is a pointer pointer, you must dereference the result to find the selector. For convenience, the Memory panel should be set to display long hexadecimal; if its not, choose Options from the Tools menu, select the Debug display, and set the format to be Long Hex. Select the first hex address associated with the selector stack address, drag it slightly within the Memory panel, and drop it. This deferences the pointer and displays the selector name.

Porting Guide

Ensuring Portablility

1. Convert NEXTSTEP 3.*x* applications to NEXTSTEP 4.0 (OPENSTEP 4.0 for Mach)
2. Locate and convert UNIX dependencies in your application code.
3. If necessary, make your application conform to Windows conventions.
4. Build the project on Mach.

OPENSTEP applications that are free of UNIX dependencies and that compile without error on OPENSTEP for Mach should also compile without error on OPENSTEP for Windows. This section describes what you must do to ensure that your OPENSTEP applications are completely portable to Windows.

Converting NEXTSTEP 3.*x* Applications

Before you can port your NEXTSTEP 3.*x* applications to Windows, you must convert them to OPENSTEP for Mach (NEXTSTEP 4.0). NeXT provides some TOPS conversion scripts to help you make these conversions. (TOPS is a tool that performs in-place substitutions on code.) You must run these scripts in the proper sequence, and complete the necessary post-processing alterations.

The *OPENSTEP Conversion Guide* describes the conversion process and gives detailed instructions for completing it. This guide is located on-line on your OPENSTEP for Mach system at **/NextLibrary/Documentation/NextDev/Conversion/ConversionGuide**. You can access the *Conversion Guide* on-line through Digital Librarian.

Removing UNIX Dependencies

When you successfully convert your application to 4.0, you are far along in the process, but you are not quite finished. Even if your application compiles without error and runs without problems as a OPENSTEP for Mach application, you still might have to complete a deep conversion before your application can compile and run problem-free on Windows. This deep conversion requires you to locate calls of UNIX library routines in your code and replace them with equivalent Windows functions or Objective-C messages.

The general procedure to follow is:

1. Identify those source code files that **#import** or **#include** UNIX header files, particularly **bsd**. To assist you, use Project Builder's Project Find panel (textual find mode), or **grep** or a similar tool.

2. Examine the files to locate where routines in these libraries are being called.

3. Replace the UNIX routine with something more portable. There are two kinds of conversions you can make:

    * Converting UNIX routines to their underscore-prefixed counterparts on Windows

    * Converting UNIX routines to Objective-C (Foundation) messages

Converting UNIX routines to the corresponding Windows routines is an easier and hence faster kind of change to make. However the implementation of these routines may differ somewhat from their UNIX counterparts. Your application will probably run fine on Windows if it calls these functions but then again, it might not. Conversion to Objective-C messages ensures greater portability, and eliminates the need to convert data to types required by arguments (for instance, NSStrings to C

strings).

Converting to Windows Run-Time Functions

Windows supports a run-time library of functions, many of which are analogous to ANSI and other common C library routines. The Windows versions of these functions usually have an underscore character prepended to their names.

To see if a UNIX function has a counterpart in a Windows library:

1. Run Visual C++.
2. Choose Help->Run-Time Routines.
3. Search the index in the Help Topics: Run-Time Routines Quick Reference panel.
4. If there is an equivalent routine, enclose the lines containing the UNIX and the Windows routines in preprocessor conditionals (for example, **#ifndef WIN32...#else...#endif**).

Converting to Objective-C Messages

The following tables cover some of the more common UNIX-to-Objective-C conversions. The OPENSTEP classes related to the suggested methods appear both before the table and in parentheses. There may be other conversions to make that you can discover for yourself by examining the documentation of these and other classes in the OpenStep specification (**/NextLibrary/Documentation/OpenStepSpec**) and in the reference documentation for the Application Kit and Foundation frameworks. You can access this documentation on Mach using Digital Librarian or through Project Builder's Project Find panel.

String and Character Manipulation

Related classes: NSString (including methods in NSPathUtilities.h), NSScanner, NSCharacterSet.

Header File	Function	Suggestions for Replacement
ansi/string.h	strcpy(), strncpy()	copy, mutableCopy
bsd/strings.h	strcat(), strncat(),	stringByAppendingString: (NSString)
	strcmp(), strncmp()	isEqualToString:, compare: (NSString)
	strlen()	length (NSString)
ansi/stdio.h	sprintf(), fprintf()	stringWithFormat: (NSString)
	sscanf(), fscanf()	see NSScanner.h or related documentation
ansi/ctype.h	is*XXX*(), to*XXX*()	see NSCharacterSet.h or related documentation

Logging and Exceptions

Related classes: NSString (including methods in NSPathUtilities.h), NSScanner, NSCharacterSet.

Header Files	Functions	Suggestions for Replacement
ansi/stdio.h	printf(), fprintf()	NSLog()
bsd/sys/printf.h	printf(), log(), panic()	NSLog(), see NSException
bsd/sys/syslog.h	openlog(), syslog()	NSLog()

File System and I/O Operations

Related classes: NSString (including methods in NSPathUtilities.h), NSFileManager

Header Files	Function/Type	Suggestions for Replacement
bsd/sys/param.h	MAXPATHLEN	Unnecessary with NSString path utilities (NSPathUtilities.h). Can be converted to FILENAME_MAX (ansi/stdio.h).
ansi/string.h bsd/strings.h	strcat(), strncat()	stringByAppendingPathComponent:, stringByAppendingPathExtension:, and others (NSString, as defined in NSPathUtilities.h)
ansi/string.h bsd/strings.h	index(), rindex()	pathExtension, stringByDeletingPathExtension:, (NSPathUtilities.h) rangeOfString:options (NSString). Also index() can be converted to strchr() and rindex() to strrchr() (ansi/string.h).
bsd/libc.h	getwd() chmod() stat(), lstat() statfs() symlink() readlink() mkdir(), umask() rmdir() creat(), open() read(), write() move/copy operations using above functions	currentDirectoryPath (NSFileManager) changeFileAttributes:atPath (NSFileManager) fileAttributesAtPath:traverseLink:(NSFileManager). If you keep stat() calls, be advised theres no S_IFLNK or S_ISVTX flags. fileSystemAttributesAtPath: (NSFileManager) Windows does not have symbolic links or hard links; you must devise your own replacement code. createDirectoryAtPath:attributes: removeFileAtPath:handler: (NSFileManager) createFileAtPath:contents:attributes: contentsAtPath: , movePath:toPath:handler:, copyPath:toPath:handler: (NSFileManager). If you keep open(), specify the O_BINARY flag.
bsd/unistd.h	getcwd() chdir() chown(), chmod() access()	currentDirectoryPath, changeCurrentDirectoryPath: (NSFileManager) changeFileAttributes:atPath (NSFileManager) fileExistsAtPath:, isReadableFileAtPath:, isWritableFileAtPath:, isExecutableFileAtPath: isDeletableFileAtPath: (NSFileManager)
bsd/sys/dir.h bsd/sys/dirent.h	opendir(), readdir()	directoryContentsAtPath:, subPathAtPath: enumeratorAtPath: (NSFileManager)
ansi/i386/limits.h	MININT, MAXINT, MINFLOAT, MAXFLOAT	INT_MIN, INT_MAX FLT_MIN, FLT_MAX

Date and Time

Related classes: NSDate, NSCalendarDate, NSTimeZone, NSTimeZoneDetail. These classes are defined in NSDate.h.

Header Files	Functions	Suggestions for Replacement
ansi/time.h	time()	date and date... variants (NSDate)

ctime(), asctime()	description:, descriptionWithCalendarFormat:locale: (NSDate)
localtime(), gmtime()	calendarDate, dateWithCalendarFormat:locale: Access time elements with dayOfMonth, hourOfDay, etc. (NSCalendarDate)
mktime()	dateWithYear:month:day:hour:minute:second:timeZone:
strftime()	descriptionWithCalendarFormat:locale: (NSDate and NSCalendarDate)
tzset()	See documentation on the NSTimeZone class.

Dynamic Memory

Related classes: Use NSData and NSMutableData as a replacement for buffered data (unstructured memory). For allocation of string objects, use the **string...** class methods of NSString.

Header Files	Functions	Suggestions for Replacement
ansi/stdlib.h bsd/sys/malloc.h	malloc() calloc() realloc()	dataWith... methods (NSData) dataWithLength: (NSMutableData) setLength: (NSMutableData)
ansi/string.h	memcpy(), memset(),	getBytes:, subdataWithRange: (NSData)
bsd/memory.h	memcmp()	isEqualToData: (NSData)
ansi/string.h	bcopy(), bcmp(), bzero()	getBytes:, subdataWithRange: (NSData) isEqualToData: (NSData) Also memcpy(), memcmp(), memset()

Miscellaneous

Header Files	Functions	Suggestions for Replacement
bsd/unistd.h	sleep(), alarm(), wait()	See NSTimer
bsd/lib.h	environ (global type)	See NSProcessInfo
mach/cthreads.h	cthread_fork(), etc.	NSThread

Conforming to Windows Conventions

In Windows applications, menus are laid out horizontally under the title bar of a window; there is no such thing in Windows as a menu separate from a standard window, as there is in OpenStep for Mach. The Windows operating system also has different conventions for the menus that appear in this menu bar and what those menus contain. Because of these differences, you might want to make your applications' menus conform to Windows conventions for menus.

Menu Guidelines

Here are a few simple guidelines to follow to make sure you're creating an application that follows conventions Windows users have come to expect:

* Usually, an application should have at least five menus:
  File
  Edit

Window
Services
Help

If the application doesn't create, open, or save files, the File menu can be given a more appropriate name. However, it must still exist because it contains the Exit command.

* If the application doesn't support having multiple documents open, remove the Window menu.

* The File menu should be the first menu unless there's a really good reason not to. (For example, for Project Builder, it's much more appropriate to have Project be the first menu.)

* The first menu, whatever it is, should always contain the Exit command.

* The Preferences command, if it exists, belongs on the Tools menu. If there is no Tools menu, place it on the first menu, directly above Exit, and put a separator line above the Preferences command.

This list summarizes standard OPENSTEP menus and menu commands and suggests where they should go on the Windows version of your application.

Main menu	**What to do**
Info menu	Delete (see below)
File menu	As-is (see below)
Edit menu	As-is
Format menu	See below
Window	See below
Print...	Move to File menu
Services menu	Keep
Hide	Delete
Quit	Change to Exit, move to first menu.
custom menus	Probably OK as-is

Info menu	**What to do**
Don't use this menu on Windows applications. Put its commands elsewhere:	
Info Panel...	Move to last command in Help menu. Change name to 'About *application_name'*
Preferences	Move to Tools menu, if one exists, or to the next-to-last command in first menu.
Help	Put Help items (Windows Help) in the Help menu.

File menu

Since this is usually the first menu, add Preferences and Exit to it. Also, add the Print and Page Setup commands to this menu.

Format menu

Rename the Page Layout command to "Page Setup" and move it to the File menu.

Windows menu

Rename the menu "Window" and remove the Miniaturize and Close commands.

For for information on the conventions for Windows menus, *see The Windows Interface Guidelines for Sofware Design*, by Microsoft Press.

Multiple document applications

Follow the Windows guidelines for multiple document applications that don't use the MDI paradigm. Basically, this means that when the application starts up, it should display a window with a menu bar (an "untitled" document). The application should quit if there is no window with a menu bar visible (as happens when users close the last window in the application.) See the Draw project in **/NextDeveloper/Examples/AppKit** for an example.

Preparing to Build the Project on Mach

Before you build your application on OPENSTEP for Mach, you have to set up the development environment so that its compatible with OPENSTEP for Windows. First, make sure that **gnumake** (version 3.74) is specified as the make utility (it is the default). To do this, you need only open your project in Project Builder on OpenStep for Mach and save it. This substitutes **gnumake** for **make** for builds thereafter. To verify this change has take place:

1. In Project Builder, click the Project Inspector.
2. Bring up the Build Options display.
3. The path in the Build Tool field should be /**bin/gnumake**.

When you build the application, fix all reported problems until the application builds error- and (preferably) warning-free.

Porting Makefile Customizations

The templates for files **Makefile.preamble** and **Makefile.postamble** for OPENSTEP 4.0 (both Windows and Mach) differ significantly from their NEXTSTEP 3.*x* predecessors. The procedure for porting makefiles on your OPENSTEP for Mach system is:

1. Rename your current **Makefile.preamble** and **Makefile.postamble** files (for example, to **Makefile.preamble.old**).

2. Copy **Makefile.preamble.template** and **Makefile.postamble.template** in **/NextDeveloper/Makefiles/project** to your project directory; remove the **.template** suffix.

3. Copy customizations in the old makefiles to the appropriate variables in the new makefiles. Carefully read the comments in the new makefiles to identify the corresponding variable. When you are finished, save the makefiles.

Be careful when making these customizations because they must now work on two operating systems. The locations and names of tools will be different. And remember, when you specify path in makefiles prepend **"$(NEXT_ROOT)"** to the pathname. However, *don't* use NEXT_ROOT when specifying install directories.

Conditional Makefile Commands

In **Makefile.preamble** and **Makefile.postamble** you can conditionally specify options or macros for building projects on Windows. Enclose the commands in an **ifeq...endif** statement in which you use the PLATFORM_OS macro to test for the underlying operating system. For example, if you are building a bundle on Windows, you'd want to ensure that the Microsoft linker is invoked with the

-force option; thus, in **Makefile.preamble**, you type:

```
ifeq ("$(PLATFORM_OS)", "winnt")
  OTHER_LDFLAGS += "-Xlinker -force"
endif
```

Porting Guide

Introduction

This document explains how to port OPENSTEP applications built for the Mach operating system to Windows. This procedure has three phases:

1. Build your application on an OPENSTEP 4.0 for Mach system, ensuring that it thoroughly conforms to the implementation of OPENSTEP on Mach and is free of UNIX dependencies.

2. Copy the project files to Windows.

3. Build the application on Windows. If there are problems, debug the application using the GNU debugger (gdb) or, for some situations, the Visual C++ debugger.

You also might find it necessary to modify the user interface and launch behavior of ported applications because of the different conventions for menus and documents on OPENSTEP for Mach and Windows. See "Conforming to Windows Conventions" for summaries of these conversions.

Note: OPENSTEP 4.0 for Mach is sometimes informally referred to as NEXTSTEP 4.0.

Overview of OPENSTEP 4.0 for Windows

OPENSTEP 4.0 for Windows is an object-oriented graphical user and development environment for Windows NT and Windows 95. It is an implementation of the OPENSTEP standard which is based on NeXT's open object layer but with extensions that mirror the implementation of OPENSTEP for Mach

With OPENSTEP for Windows you'll be able to develop object-oriented, three-tier client-server applications in a fraction of the time it takes with current Windows development tools. These applications look and behave like any other Windows application because they are displayed inside Windows windows and use Windows-style menus. Yet they retain much of the distinctive look and feel they would have under OPENSTEP 4.0 for Mach.

You can deploy OPENSTEP for Windows applications on Intel-based PCs running the OPENSTEP for Windows run-time system and easily port them across all OPENSTEP implementations, including those announced from Sun, Hewlett Packard, and NeXT. Your applications will seamlessly communicate with objects running on servers from Sun, Hewlett Packard and Digital; they will also automatically communicate with OLE objects and services, enabling OPENSTEP applications to interoperate with native Windows applications such as Excel and Word.

NEXT_ROOT

When you install OPENSTEP for Windows, you identify the root directory for the software through the user environment variable NEXT_ROOT. References to paths (in makefiles, for instance) are thus often prefixed with **"$(NEXT_ROOT)"**.

Major Components

Before you begin porting your application, you might find it useful to know what the major components of OPENSTEP are and where they're located on your NT system. This section

describes those components.

Development Applications

Both Interface Builder and Project Builder have been ported to Windows, so you can use both in application development. Interface Builder and Project Builder are located in **NEXT_ROOT/NextDeveloper/Apps**.

The following tables describe the other important components.

Run Time

The following processes provide the run-time support needed for OpenStep for Windows. In the installation procedure you can request Windows to start up these processes when the system is booted. These processes are located in **NEXT_ROOT/NextLibrary/System**.

Process	Description
machd.exe	Mach emulator
nmserver.exe	Mach network server
WindowServer.exe	Display PostScript window server
pbs.exe	pasteboard server

Tools

The compiler and the make utitlity are in **NEXT_ROOT/NextDeveloper/Executables**. The Bourne shell and the shell utilities are in **NEXT_ROOT/NextLibrary Executables**.

Tool	Description
gcc.exe	Gnu C/C++ compiler driver. Executable files called by **gcc** (including **as**, the assembler) are in **NEXT_ROOT/NextDeveloper/Libraries/gcc-lib/winnt3.5/2.6.1**.
as.exe	Gnu assembler
make.exe	Gnu make utility (3.74)
sh.exe	Bourne shell 4.4 BSD
shell utilities	UNIX-style command line utilities (**grep**, **ls**, **chmod**, **vi**, **less**, etc.)

Note: The link editor, **ld**, has not been ported. For the current release, the compiler driver **gcc** automatically uses **link.exe**, the Visual C++ linker.

Frameworks

Frameworks are located in **NEXT_ROOT/NextLibrary/Frameworks**. Related DLLs are in **NEXT_ROOT/NextLibrary/Executables**.

Framework	Description
AppKit.framework	Contains a library of Application Kit classes, protocols, functions, and types along with associated header files, documentation, and resources.
Foundation.framework	Contains a library of Foundation Kit classes, protocols, functions, and types, along with associated header files, documentation, and resources
EOInterface.framework EOAccess.framework EOControl.framework Oracle.framework	Frameworks related to the Enterprise Objects Framework (EOF) version 2.0. They contain libraries of classes, protocols, functions, and types, along with associated header files, documentation, and resources. Included are frameworks for the access layer, the interface layer, the control layer and for

Sybase.framework Informix.framework	various database adaptors. The “1x” frameworks (for example, EOInterface1x.framework) are version 1.2 frameworks.