Porting UNIX/Linux Applications to OS X

Developer

Contents

Figures, Tables, and Listings

Introduction to Porting UNIX/Linux Applications to OS X

The UNIX Porting Guide is a first stop for UNIX developers coming to OS X. This document helps guide developers in bringing applications written for UNIX-based operating systems to OS X. It provides the background needed to understand the operating system. It touches on some of the design decisions, and it provides a listing and discussion of some of the main areas that you should be concerned with in bringing UNIX applications to OS X. It also points out some of the advanced features of OS X not available in traditional UNIX applications that you can add to your ported applications.

This document also provides an entry point for other documentation on various subjects that may be of interest if you are porting an application from a UNIX environment to OS X.

This document is an overview, not a tutorial. In many regards it is a companion to the more extensive *Mac Technology Overview* , but with a bias toward the UNIX developer.

This document also does not cover porting shell scripts to OS X. For more information about shell scripts and OS X, you should read *Shell Scripting Primer* .

Bringing UNIX Apps to OS X

The introduction of UNIX-like operating systems such as FreeBSD and Linux for personal computers was a great step in bringing the power and stability of UNIX to the mass market. Generally though, these projects were driven by power users and developers for their own use, without making design decisions that would make UNIX palatable to consumers. OS X, on the other hand, was designed from the beginning with end users in mind.

With this operating system, Apple builds its well-known strengths in simplicity and elegance of design on a UNIX-based foundation. Rather than reinventing what has already been done well, Apple is combining their strengths with the strengths brought about by many years of advancement by the UNIX community.

Who Should Read This Document?

Any UNIX developers can benefit from reading this book.

- In-house corporate application developers

- Commercial UNIX developers
- Open source developers
- Open source porters
- Higher education faculty, staff, and students
- Science and research developers

If you're a commercial UNIX developer, you are already familiar with other UNIX-based systems and may want to understand the differences between other systems and OS X. You might be interested in porting the GUI from an X11 environment into a native graphics environment using Carbon or Cocoa. You may also have special needs such as direct hardware access, exclusive file access guarantees, and so on.

If you're a corporate in-house developer (developing applications for internal use), you probably want to port applications with minimal code divergence.

If you're an open source developer, you might want information about how to incorporate new technologies into your software, and may be interested in GUI porting, depending on your level of interest. Alternately, you might be interested only in quickly porting code to a new platform with minimal changes so that you can easily get your changes back into the official code base. If so, you may be more likely to use compatibility shims than to use new APIs.

No matter what "flavor" of developer you are, this book will provide information that is helpful to you and provide pointers to additional documents that may be of interest.

Important: If you are primarily interested in shell scripts and command-line compatibility, you should read "Designing Scripts for Cross-Platform Deployment" in *Shell Scripting Primer* . That document gives a more thorough overview of the shell environment in OS X, including common cross-platform compatibility issues.

Important: This document is not designed for pure Java developers. OS X has a full and robust Java 2 Platform, Standard Edition (J2SE) implementation. If you have a pure Java application already, it should run in OS X.

Organization of This Document

This document is a first stop for UNIX developers coming to OS X. It contains many links to more extensive documentation. Specific details of implementation are covered here only in cases where it is not adequately covered in other places in the documentation set.

To use this document most effectively, first read "Overview of OS X" (page 10) to find out the basics about the Mac and to get some of the high-level information you need to begin your port. If you already have an application that builds on other UNIX-based platforms, "Compiling Your Code in OS X" (page 19) will help you find out how to compile your code on OS X.

Most of your effort, however, should be spent towards making decisions concerning which, if any, graphical user interface to implement with your application. "Choosing a Graphical Environment for Your Application" (page 49) helps you with this.

If you want to refactor your application to take advantage of the rich feature set of OS X, see "Additional Features" (page 68) for examples of features available in OS X.

Once you have a complete application, read "Distributing Your Application" (page 64) for information on getting your application to OS X users.

See Also

Developer documentation can be found at Apple's developer website at http://developer.apple.com/. This site contains reference, conceptual, and tutorial material for the many facets of development on OS X. The OS X Developer Tools CD includes a snapshot of the developer documentation, which can be searched for and viewed in Xcode's doc viewer. The `man` pages are also included with the OS X Developer Tools.

Apple Developer Connection (ADC) offers a variety of membership levels to help you in your development. These range from free memberships that give you access to developer software, to paid memberships that provide support incidents as well as the possibility of software seeds. More information on memberships is available at http://developer.apple.com.

Once a year in early Summer, Apple hosts the Worldwide Developers Conference (WWDC) in the San Francisco, California Bay area. This is an extremely valuable resource for developers trying to get an overall picture of OS X as well as specific implementation details of individual technologies. Information on WWDC is available on the ADC website.

Apple hosts an extensive array of public mailing lists. These are available for public subscription and searching at http://lists.apple.com. The *unix-porting* list is highly recommended. The *darwin-dev* and *darwinos-users* lists also offer much help but less specific to the task of porting.

In addition to Apple's own resources, many external resources exist, for example, O'Reilly's Mac DevCenter, http://www.oreillynet.com/mac/.

Overview of OS X

OS X is a modern operating system that combines the power and stability of UNIX-based operating systems with the simplicity and elegance of the Macintosh. For years, power users and developers have recognized the strengths of UNIX and its offshoots. While UNIX-based operating systems are indispensable to developers and power users, consumers have rarely been able to enjoy their benefits because of the perceived complexity. Instead consumers have lived with a generation of desktop computers that could only hope to achieve the strengths that UNIX-based operating systems have had from the beginning.

This chapter is for anyone interested in an overview of OS X—its lineage and its open source core, called Darwin. Here you will find background information about OS X and how your application fits in.

The Family Tree

Although this document covers the basic concepts in bringing UNIX applications to OS X, it is by no means comprehensive. This section is provided to give you a hint on where to look for additional documentation by outlining how OS X came to be. Knowing a little about the lineage of OS X will help you to find more resources as the need arises.

• BSD

Part of the history of OS X goes back to Berkeley Software Distributions (BSD) UNIX of the late seventies and early eighties. Specifically, it is based in part on BSD 4.4 Lite. On a system level, many of the design decisions are made to align with BSD-style UNIX systems. Most libraries and utilities are from FreeBSD (http://www.freebsd.org/), but some are derived from NetBSD (http://www.netbsd.org/). For future development, OS X has adopted FreeBSD as a reference code base for BSD technology. Work is ongoing to more closely synchronize all BSD tools and libraries with the FreeBSD-stable branch..

• Mach

Although OS X must credit BSD for most of the underlying levels of the operating system, OS X also owes a major debt to Mach. The kernel is heavily influenced in its design philosophy by Carnegie Mellon's Mach project. The kernel is not a pure microkernel implementation, since the address space is shared with the BSD portion of the kernel and the I/O Kit.

• NEXTSTEP

In figuring out what makes OS X tick, it is important to recognize the influences of NEXTSTEP and OPENSTEP in its design. Apple's acquisition of NeXT in 1997 was a major key in bringing OS X from the drawing board into reality. Many parts of OS X of interest to UNIX developers are enhancements and offshoots of the technology present in NEXTSTEP. From the file system layer to the executable format and from the high-level Cocoa API to the kernel itself, the lineage of OS X as a descendant of NEXTSTEP is evident.

• Earlier Version of the Mac OS

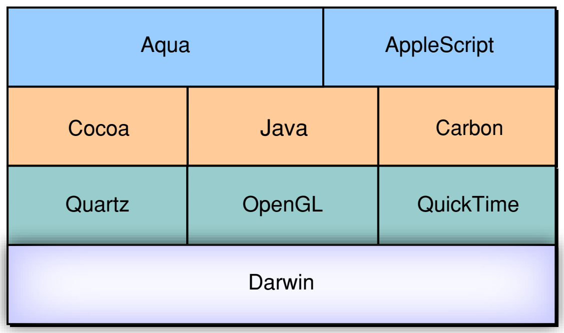
Although it shares its name with earlier versions of the Mac OS, OS X is a fundamentally new operating system. This does not mean that all that went before has been left out. OS X still includes many of the features that Mac OS 9 and earlier versions included. Although your initial port to OS X may not use any of the features inherited from Mac OS 9, as you enhance the application, you might take advantage of some of the features provided by technologies like ColorSync or the Carbon APIs. Mac OS 9 is also the source of much of the terminology used in OS X.

OS X and Darwin

The word *Darwin* is often used to refer to the underpinnings of OS X. In fact, in some circles OS X itself is rarely mentioned at all. It is important to understand the distinction between the two—how they are related and how they differ.

Darwin is the core of the OS X operating system. Although Darwin can stand alone as an independent operating system, it includes only a subset of the features available in OS X. Figure 1-1 (page 11) shows how Darwin is related to OS X as a whole.

Figure 1-1 Darwin's relation to OS X



Darwin is an open source project. With it, you as a developer gain access to the foundation of OS X. Its openness also allows you to submit changes that you think should be reflected in OS X as a whole. Darwin has been released as a separate project that runs on PowerPC-based Macintosh computers as well as x86-compatible computers. Although it could be considered a standalone operating system in its own right, many of the fundamental design decisions of Darwin are governed by its being embedded within OS X. In bringing your applications to the platform, you should target OS X version 10.1.4 (Darwin 5.4) or later.

OS X itself is not an Open Source project. As you can see from Figure 1-1 (page 11), there are many parts of OS X that are not included in the Open Source Darwin components. Part of your job while porting is deciding where your application will fit in OS X.

If you are a developer whose tool is a command-line tool (or has a useful subset that is a command-line tool), you can, of course, simply port your application as a command-line tool or service, which is usually not that complicated. By doing this you gain a small benefit, in that it is now available to OS X users who are familiar with the UNIX command-line environment. You will not be able to market it to OS X users as a whole though, since many users do not even know how to access the command line on their computers.

The basic steps in porting a UNIX application to OS X typically include:

1. Port to the command line.
2. Provide a graphical user interface (GUI).

What Macintosh Users Expect

In bringing your UNIX application to OS X, you are entering a world where great emphasis is placed on user interactions. This brings you many opportunities and benefits as a developer, but also some responsibilities.

Benefits of Porting to OS X

Porting to OS X has three primary benefits:

- stable long-term customer base
- good inroad into education
- powerful developer tools

Bringing UNIX applications to OS X can be very profitable if done correctly. Well-designed Macintosh applications of years past are the standards of today. PhotoShop, Illustrator, and Excel are all applications that first made their name on the Macintosh. Now is the time to win the hearts of Macintosh users with the next great application. In a word, possibly millions of paying customers!

Macintosh users are willing to spend their money on great applications because they know that Apple strives to give them a high-quality user environment. Apple developers are known for providing great applications for that environment.

For years, Apple has been known for its commitment to education. OS X targets the education market for developers and is an ideal platform for learning for students. With its standards-based technologies as well as home-grown technologies, you have an ideal platform for use in educational application deployment and development.

OS X also provides benefits in a development environment. Apple strives for standards first, then it adds that little bit that makes it better on a Mac. As a developer, you have access to many of the development tools and environments that you have on other platforms, like Java, OpenGL, POSIX libraries, and the BSD TCP/IP stack, but you also have built-in benefits like the Apache Web server on every computer, the Cocoa object-oriented development environment, a PDF-based display system (Quartz), Kerberos, QuickTime, a dynamic core audio implementation, and a suite of world-class developer tools. By adding a native OS X front end to your application, you can achieve a cost-effective new deployment platform with minimal additional development effort.

OS X adds tremendous value both to you and your customers on a standards-based operating system.

Responsibilities of Porting to OS X

Along with benefits come responsibilities. If you have decided to make a full-featured Mac app, here are some guidelines to keep in mind.

- An OS X user should never have to resort to the command line to perform any task in an application with a graphical user interface. This is especially important to remember since the BSD user environment may not even be installed on a user's system. The libraries and kernel environment are of course there by default, but the tools may not be.
- If you are making graphical design decisions, you need to become familiar with the *OS X Human Interface Guidelines* , available from the Apple developer website. These are the standards that Macintosh users expect their applications to live up to. Well-behaved applications from Apple and third-party developers give the Macintosh its reputation as the most usable interface on the planet.

The responsibilities boil down to striving for an excellent product from a user's perspective. OS X gives you the tools to make your applications shine.

Preparing to Port

A seasoned UNIX developer recognizes that no matter how similar two UNIX-based operating systems may be, there are always details that set one apart from another.

This chapter highlights areas to be aware of when compiling your code base for OS X. It notes details about compiler flags that are important to you, and gives insight into how to link different parts of your code in OS X. Many of these topics are covered more extensively in other resources as noted.

With few exceptions, this chapter applies to all varieties of UNIX developers. However, there are a few things that are specific to a given audience. These are pointed out as they appear.

If you are porting an open source application, you should first check to see if any of the open source port collections have already ported the application to avoid duplicating effort. You should also consider including your port in one of these collections to make it easier for others to take advantage of it. Finally, you should always make your changes available upstream to the original open source project for inclusion in future releases.

Some of the Darwin-based open source port collections include:

- MacPorts (http://www.macports.org/)
- Fink (http://fink.sourceforge.net/)
- GNU-Darwin (http://www.gnu-darwin.org/)

For more information on these port collections, visit their websites.

Important: This information is provided solely for your convenience, and that mention of these projects in no way represents any recommendation or endorsement by Apple.

Before you bring the basic port of your code to OS X, make sure that you have the requisite tools for the task. It helps to know what is and isn't available to you by default.

The OS X Command-Line Environment

The most important thing to do before porting an application is to familiarize yourself with the OS X environment. In particular, you should familiarize yourself with the System Preferences application (and customize your computer to suit) and the Terminal application (to obtain a command-line interface).

The System Preferences application is located in the `Applications` folder at the root level of your hard drive. The `Terminal` application is located in the `Utilities` folder (which is within the `Applications` folder at the root level of your hard drive).

Once you have a Terminal window open, you can take advantage of a basic selection of common tools, assuming that they are installed. Before using OS X as a development platform, you should make sure that the BSD subsystem is installed. On OS X v10.4, this option is installed by default, but on previous versions, it was not.

You can check for this by looking for the BSD package receipt, `BSD.pkg`, in `/Library/Receipts`. If this receipt is not present on your system, you must reinstall OS X. When you do, you must customize the installation by checking the BSD Subsystem checkbox.

> **Note:** If you have already installed software updates, you cannot simply go back and reinstall over the existing system because of the potential for version conflicts with the software updates. If the BSD subsystem package is not installed, you will have to do an "archive and install" installation, then reinstall the software updates.

With the BSD subsystem installed, a look through `/bin` and `/usr/bin` should reveal a familiar environment. Welcome home.

Installing the OS X Developer Tools

Because OS X has a BSD core, you have access to the numerous open source tools that you are already familiar with (like the GNU tools, for example).

The OS X Xcode Tools package provides additional tools that you will need to install to round out your development environment. These are not part of the default installation, but are essential to you. They contain some of the most important tools, such as the compiler (gcc) and debugger (gdb).

Most (if not all) of the tools described in this document are part of the Xcode Tools installation, so if you don't find `autoconf` or `make`, for example, that probably means you don't have Xcode Tools installed.

Important: If you are using makefiles for compilation, you should install Xcode in the default location (`/usr`). If you do not, you may have to do extra work to get your scripts to run the compiler, linker, and so on in a nonstandard location.

If you are compiling UNIX applications, be sure to select the optional BSD SDK package when installing Xcode tools. This tells the installer to install a number of BSD-level headers, without which many UNIX applications will not compile.

You can find the Xcode Tools Installer on your OS X DVD (or as one of the CDs in the CD version). In addition, you can always obtain the very latest version online from the Apple Developer Connection (ADC) website, http://connect.apple.com or the app store. You need an ADC account to download the Developer Tools. Free accounts are available for those who just need access to the tools.

After installing the OS X Developer Tools, you will have a selection of new tools to take advantage of:

- `/usr/bin` now contains more command-line tools than were supplied by the BSD package alone, most notably gcc and gdb. `/usr/bin/cc` defaults to gcc 3 in OS X version 10.2 and later, but gcc 2.95.2 is also available. See "Choosing a Compiler " (page 26) for more information.
- Xcode is a graphical integrated development environment for applications in multiple programming languages.
- Interface Builder provides a simple way to build complex user interfaces.
- FileMerge lets you graphically compare and merge files and directories.
- IORegistryExplorer helps you determine which devices are registered with the I/O KIT. See "Device Drivers" (page 70) for a discussion of the I/O Kit.
- MallocDebug analyzes all allocated memory in an application. It can measure the memory allocated since a given point in time and detect memory leaks
- PackageMaker builds easily distributable OS X packages.

Documentation for these tools is available from the Apple Technical Publications website.

Building Makefile Projects With Xcode

OS X-native applications can be built in a number of environments. Most UNIX utilities are built using makefiles, though some are built using build scripts, imakefiles, or various other mechanisms.

If you are porting a UNIX command-line tool for personal use, you probably want to continue using the existing build environment, because keeping multiple build environments synchronized can be tricky at best. However if you are porting a more extensive tool and you plan to add an OS X GUI to the tool, you may find it more convenient to work with the project in a development environment such as Xcode.

Note: These instructions apply to the latest version of Xcode.

Although Xcode keeps track of build settings in its own preferences files for information beyond what could normally be maintained in a makefile, it can also work closely with your project's makefiles. If you want to use Xcode for development in OS X, you can include a makefile in a Xcode project as follows:

1. Launch Xcode.
2. Choose New Project from the File menu.
3. Select whatever project type you are targeting. If you ultimately want an application, select something like Cocoa Application. If you are just trying to build a command-line utility, select one of the tools—for example, Standard Tool.
4. Follow the prompts to name and save your project. A new default project of that type is opened.
5. Open the Targets disclosure triangle and delete any default targets that may exist.
6. From the Project menu, Choose New Target.
7. Select "External Target" from the list. If this is not shown in the "Special Targets" list, you are not running the latest version of Xcode. Upgrade first.
8. Follow the prompts to name that target. When you have done this, a target icon with the name you just gave it appears in the Targets pane of the open Xcode window.
9. Double-click that new target. You should now see a new window with the build information for this target. This is *not* the same thing as clicking info. You *must* double-click the target itself.
10. In the "Custom Build Command" section of the target inspector, change the field called "Directory" to point to the directory containing your makefile, and change any other settings as needed. For example, in the Custom Build Settings pane, you could change Build Tool from `/usr/bin/gnumake` to `/usr/bin/bsdmake`. More information on the fields is available in Xcode Help.
11. Change the active target to your new target by choosing "Set Active Target" from the Project menu.
12. Add the source files to the project. To do this, first open the disclosure triangle beside the "Source" folder in the left side of the project window. Next, drag the folder containing the sources from the Finder into that "Source" folder in Xcode. Tell Xcode not to copy files. Xcode will recursively find all of the files in that folder. Delete anything you don't want listed.
13. When you are ready to build the project, click the Build and Run button in the toolbar, select Build from the Build menu, or just press Command-B.

14. Once the project is built, tell Xcode where to find the executable by choosing "New Custom Executable" from the Project menu. Choose the path where the executable is located, then add the name of the executable.
15. Run the resulting program by pressing Command-R.

This should get you started in bringing your application into the native build environment of OS X.

Windowing Environment Considerations

Before you compile an application in OS X, be aware that the OS X native windowing and display subsystem, Quartz, is based on the Portable Document Format (PDF). Quartz consists of a lightweight window server as well as a graphics rendering library for two-dimensional shapes. The window server features device-independent color and pixel depth, layered compositing, and buffered windows. The rendering model is PDF based.

Quartz is not an X Window System implementation. If you need an X11R6 implementation, you can easily install one. For more information, see "X11R6" (page 54).

Information on the graphical environments available to you can be found in "Choosing a Graphical Environment for Your Application" (page 49) along with some things to consider when choosing a graphical environment.

Working with 64-bit Software

Beginning in version 10.4, OS X supports execution of 64-bit command-line tools. It is also possible to use a 32-bit OS X front-end application to provide a user interface through careful use of client-server communication.

This document primarily covers general porting issues. Issues specific to 64-bit computing are beyond the scope of this document. For additional information on porting 64-bit software to OS X, see the document *64-Bit Transition Guide* .

Compiling Your Code in OS X

Now that you have the basic pieces in place, it is time to build your application. This section covers some of the more common issues that you may encounter in bringing your UNIX application to OS X. These issues apply largely without regard to what type of development you are doing.

Using GNU Autoconf, Automake, and Autoheader

If you are bringing a preexisting command-line utility to OS X that uses GNU `autoconf`, `automake`, or `autoheader`, you will probably find that it configures itself without modification (though the resulting configuration may be insufficient). Just run `configure` and `make` as you would on any other UNIX-based system.

If running the `configure` script fails because it doesn't understand the architecture, try replacing the project's `config.sub` and `config.guess` files with those available in `/usr/share/automake-1.6`. If you are distributing applications that use autoconf, you should include an up-to-date version of `config.sub` and `config.guess` so that OS X users don't have to do anything extra to build your project.

If that still fails, you may need to run `/usr/bin/autoconf` on your project to rebuild the `configure` script before it works. OS X includes autoconf in the BSD tools package. Beyond these basics, if the project does not build, you may need to modify your makefile using some of the tips provided in the following sections. After you do that, more extensive refactoring may be required.

Some programs may use autoconf macros that are not supported by the version of autoconf that shipped with OS X. Because autoconf changes periodically, you may actually need to get a new version of autoconf if you need to build the very latest sources for some projects. In general, most projects include a prebuilt `configure` script with releases, so this is usually not necessary unless you are building an open source project using sources obtained from CVS or from a daily source snapshot.

However, if you find it necessary to upgrade autoconf, you can get a current version from http://www.gnu.org/software/autoconf/. Note that autoconf, by default, installs in `/usr/local/`, so you may need to modify your `PATH` environment variable to use the newly updated version. Do *not* attempt to replace the version installed in `/usr/`.

For additional information about using the GNU autotoolset, see http://autotoolset.sourceforge.net/tutorial.html and the manual pages `autoconf`, `automake`, and `autoheader`.

Compiling for Multiple CPU Architectures

Because the Macintosh platform includes more than one processor family, it is often important to compile software for multiple processor architectures. For example, libraries should generally be compiled as universal binaries even if you are exclusively targeting an Intel-based Macintosh computer, as your library may be used by a PowerPC binary running under Rosetta. For executables, if you plan to distribute compiled versions, you should generally create universal binaries for convenience.

When compiling programs for architectures other than your default host architecture, such as compiling for a ppc64 or Intel-based Macintosh target on a PowerPC-based build host, there are a few common problems that you may run into. Most of these problems result from one of the following mistakes:

- Assuming that the build host is architecturally similar to the target architecture and will thus be capable of executing intermediate build products
- Trying to determine target-processor-specific information at configuration time (by compiling and executing small code snippets) rather than at compile time (using macro tests) or execution time (for example, by using conditional byte swap functions)

Whenever cross-compiling occurs, extra care must be taken to ensure that the target architecture is detected correctly. This is particularly an issue when generating a binary containing object code for more than one architecture.

In many cases, binaries containing object code for more than one architecture can be generated simply by running the normal configuration script, then overriding the architecture flags at compile time.

For example, you might run

```
./configure
```

followed by

```
make CFLAGS="-isysroot /Developer/SDKs/MacOSX10.4u.sdk -arch ppc \
    -arch i386" LDFLAGS="-syslibroot /Developer/SDKs/MacOSX10.4u.sdk \
    -arch ppc -arch i386"
```

to generate a universal binary (for Intel-based and PowerPC-based Macintosh computers). To generate a 4-way universal binary that includes 64-bit versions, you would add `-arch ppc64` and `-arch x86_64` to the `CFLAGS` and `LDFLAGS`.

Note: If you are using an older version of gcc and your makefile passes `LDFLAGS` to gcc instead of passing them directly to ld, you may need to specify the linker flags as `-Wl,-syslibroot,/Developer/SDKs/MacOSX10.4u.sdk`. This tells the compiler to pass the unknown flags to the linker without interpreting them. Do not pass the `LDFLAGS` in this form to ld, however; ld does not currently support the `-Wl` syntax.

If you need to support an older version of gcc and your makefile passes `LDFLAGS` to both gcc and ld, you may need to modify it to pass this argument in different forms, depending on which tool is being used. Fortunately, these cases are rare; most makefiles either pass `LDFLAGS` to gcc or ld, but not both. Newer versions of gcc support `-syslibroot` directly.

If your makefile does not explicitly pass the contents of `LDFLAGS` to `gcc` or `ld`, they may still be passed to one or the other by a make rule. If you are using the standard built-in make rules, the contents of `LDFLAGS` are passed directly to `ld`. If in doubt, assume that it is passed to `ld`. If you get an invalid flag error, you guessed incorrectly.

If your makefile uses gcc to run the linker instead of invoking it directly, you *must* specify a list of target architectures to link when working with universal binary object (.o) files even if you are not using all of the architectures of the object file. If you don't, you will not create a universal binary, and you may also get a linker error. For more information about 64-bit executables, see *64-Bit Transition Guide*.

However, applications that make configuration-time decisions about the size of data structures will generally fail to build correctly in such an environment (since those sizes may need to be different depending on whether the compiler is executing a `ppc` pass, a `ppc64` pass, or an `i386` pass). When this happens, the tool must be configured and compiled for each architecture as separate executables, then glued together manually using `lipo`.

In rare cases, software not written with cross-compilation in mind will make configure-time decisions by executing code on the build host. In these cases, you will have to manually alter either the configuration scripts or the resulting headers to be appropriate for the actual target architecture (rather than the build architecture). In some cases, this can be solved by telling the configure script that you are cross-compiling using the `--host`, `--build`, and `--target` flags. However, this may simply result in defaults for the target platform being inserted, which doesn't really solve the problem.

The best fix is to replace configure-time detection of endianness, data type sizes, and so on with compile-time or run-time detection. For example, instead of testing the architecture for endianness to obtain consistent byte order in a file, you should do one of the following:

- Use C preprocessor macros like `__BIG_ENDIAN__` and `__LITTLE_ENDIAN__` to test endianness at compile time.

- Use functions like `htonl`, `htons`, `ntohl`, and `ntohs` to guarantee a big-endian representation on any architecture.
- Extract individual bytes by bitwise masking and shifting (for example, `lowbyte=word & 0xff; nextbyte = (word >> 8) & 0xff;` and so on).

Similarly, instead of performing elaborate tests to determine whether to use `int` or `long` for a 4-byte piece of data, you should simply use a standard sized type such as `uint32_t`.

Note: Not all script execution is incompatible with cross-compiling. A number of open source tools (GTK, for example) use script execution to determine the presence or absence of libraries, determine their versions and locations, and so on.

In those cases, you must be certain that the info script associated with the universal binary installation (or the target platform installation if you are strictly cross-compiling) is the one that executes during the configuration process, rather than the info script associated with an installation specific to your host architecture.

There are a few other caveats when working with universal binaries:

- The library archive utility, `ar`, cannot work with libraries containing code for more than one architecture (or single-architecture libraries generated with `lipo`) after ranlib has added a table of contents to them. Thus, if you need to add additional object files to a library, you must keep a separate copy without a TOC.
- The `-M` switch to gcc (to output dependency information) is not supported when multiple architectures are specified on the command line. Depending on your makefile, this may require substantial changes to your makefile rules. For autoconf-based configure scripts, the flag `--disable-dependency-tracking` should solve this problem.

  For projects using automake, it may be necessary to run automake with the `-i` flag to disable dependency checks or put `no-dependencies` in the `AUTOMAKE_OPTIONS` variable in each Makefile.am file.
- If you run into problems building a universal binary for an open source tool, the first thing you should do is to get the latest version of the source code. This does two things:
  - Ensures that the version of autoconf and automake used to generate the configuration scripts is reasonably current, reducing the likelihood of build failures, execution failures, backwards or forwards compatibility problems, and other idiosyncratic or downright broken behavior.
  - Reduces the likelihood of building a version of an open source tool that contains known security holes or other serious bugs.

- Older versions of autoconf do not handle the case where `--target`, `--host`, and `--build` are not handled gracefully. Different versions also behave differently when you specify only one or two of these flags. Thus, you should always specify all three of these options if you are running an autoconf-generated configure script with intent to cross-compile.
- Some earlier versions of autoconf handle cross-compiling poorly. If your tool contains a configure script generated by an early autoconf, you may be able to significantly improve things by replacing some of the `config.*` files (and `config.guess` in particular) with updated copies from the version of autoconf that comes with OS X.

  This will not always work, however, in which case it may be necessary to actually regenerate the configure script by running autoconf. To do this, simply change into the root directory of the project and run `/usr/bin/autoconf`. It will automatically detect and use the `configure.in` file and use it to generate a new configure script. If you get warnings, you should first try a web search for the error message, as someone else may have already run into the problem (possibly on a different tool) and found a solution.

  If you get errors about missing `AC_` macros, you may need to download a copy of libraries on which your tool depends and copy their `.m4` autoconf configuration files into `/usr/share/autoconf`. Alternately, you can add the macros to the file `acinclude.m4` in your project's main directory and autoconf should automatically pick up those macros.

  You may, in some cases, need to rerun `automake` and/or `autoheader` if your tool uses them. Be prepared to run into missing `AM_` and `AH_` macros if you do, however. Because of the added risk of missing macros, this should generally only be done if running autoconf by itself does not correct a build problem.

  Important: Be sure to make a backup copy of the original scripts, headers, and other generated files (or, ideally, the entire project directory) before running autoheader or automake.

- Different makefiles and configure scripts handle command-line overrides in different ways. The most consistent way to force these overrides is to specify them prior to the command. For example:

  ```
  make CFLAGS="-isysroot /Developer/SDKs/MacOSX10.4u.sdk -arch i386 -arch
  ppc"
  ```

  should generally result in the above being added to `CFLAGS` during compilation. However, this behavior is not completely consistent across makefiles from different projects.

For additional information about autoconf, automake, and autoheader, you can view the autoconf documentation at http://www.gnu.org/software/autoconf/manual/index.html.

For additional information on compiler flags for Intel-based Macintosh computers, modifying code to support little-endian CPUs, and other porting concerns, you should read *Universal Binary Programming Guidelines, Second Edition*, available from the ADC Reference Library.

Cross-Compiling a Self-Bootstrapping Tool

Probably the most difficult situation you may experience is that of a self-bootstrapping tool—a tool that uses a (possibly stripped-down) copy of itself to either compile the final version of itself or to construct support files or libraries. Some examples include TeX, Perl, and gcc.

Ideally, you should be able to build the executable as a universal binary in a single build pass. If that is possible, everything "just works", since the universal binary can execute on the host. However, this is not always possible. If you have to cross-compile and glue the pieces together with `lipo`, this obviously will not work.

If the build system is written well, the tool will bootstrap itself by building a version compiled for the host, then use that to build the pieces for the target, and finally compile a version of the binary for the target. In that case, you should not have to do anything special for the build to succeed.

In some cases, however, it is not possible to simultaneously compile for multiple architectures and the build system wasn't designed for cross-compiling. In those cases, the recommended solution is to pre-install a version of the tool for the host architecture, then modify the build scripts to rename the target's intermediate copy of the tool and copy the host's copy in place of that intermediate build product (for example, `mv miniperl miniperl-target; cp /usr/bin/perl miniperl`).

By doing this, later parts of the build script will execute the version of the tool built for the host architecture. Assuming there are no architecture dependencies in the dependent tools or support files, they should build correctly using the host's copy of the tool. Once the dependent build is complete, you should swap back in the original target copy in the final build phase. The trick is in figuring out when to have each copy in place.

Conditional Compilation on OS X

You will sometimes find it necessary to use conditional compilation to make your code behave differently depending on whether certain functionality is available.

Older code sometimes used conditional statements like `#ifdef __MACH__` or `#ifdef __APPLE__` to try to determine whether it was being compiled on OS X or not. While this seems appealing as a quick way of getting ported, it ultimately causes more work in the long run. For example, if you make the assumption that a particular function does not exist in OS X and conditionally replace it with your own version that implements the same functionality as a wrapper around a different API, your application may no longer compile or may be less efficient if Apple adds that function in a later version.

Apart from displaying or using the name of the OS for some reason (which you can more portably obtain from the `uname` API), code should never behave differently on OS X merely because it is running on OS X. Code should behave differently because OS X behaves differently in some way—offering an additional feature, not offering functionality specific to another operating system, and so on. Thus, for maximum portability and maintainability, you should focus on that difference and make the conditional compilation dependent upon detecting the *difference* rather than dependent upon the OS itself. This not only makes it easier to maintain your code as OS X evolves, but also makes it easier to port your code to other platforms that may support different but overlapping feature sets.

The most common reasons you might want to use such conditional statements are attempts to detect differences in:

- processor architecture
- byte order
- file system case sensitivity
- other file system properties
- compiler, linker, or toolchain differences
- availability of application frameworks
- availability of header files
- support for a function or feature

Instead it is better to figure out *why* your code needs to behave differently in OS X, then use conditional compilation techniques that are appropriate for the *actual* root cause.

The misuse of these conditionals often causes problems. For example, if you assume that certain frameworks are present if those macros are defined, you might get compile failures when building a 64-bit executable. If you instead test for the availability of the framework, you might be able to fall back on an alternative mechanism such as X11, or you might skip building the graphical portions of the application entirely.

For example, OS X provides preprocessor macros to determine the CPU architecture and byte order. These include:

- `__i386__`—Intel (32-bit)
- `__x86_64__`—Intel (64-bit)
- `__ppc__`—PowerPC (32-bit)
- `__ppc64__`—PowerPC (64-bit)
- `__BIG_ENDIAN__`—Big endian CPU
- `__LITTLE_ENDIAN__`—Little endian CPU
- `__LP64__`—The LP64 (64-bit) data model

In addition, using tools like `autoconf`, you can create arbitrary conditional compilation on nearly any practical feature of the installation, from testing to see if a file exists to seeing if you can successfully compile a piece of code.

For example, if a portion of your project requires a particular application framework, you can compile a small test program whose `main` function calls a function in that framework. If the test program compiles and links successfully, the application framework is present for the specified CPU architecture.

You can even use this technique to determine whether to include workarounds for known bugs in Apple or third-party libraries and frameworks, either by testing the versions of those frameworks or by providing a test case that reproduces the bug and checking the results.

For example, in OS X, `poll` does not support device files such as `/dev/tty`. If you just avoid `poll` if your code is running on OS X, you are making two assumptions that you should not make:

- You are assuming that what you are doing will always be unsupported. OS X is an evolving operating system that adds new features on a regular basis, so this is not necessarily a valid assumption.
- You are assuming that OS X is the only platform that does not support using `poll` on device files. While this is probably true for most device files, not all device files support `poll` in all operating systems, so this is also not necessarily a valid assumption.

A better solution is to use a configuration-time test that tries to use `poll` on a device file, and if the call fails, disables the use of `poll`. If using `poll` provides some significant advantage, it may be better to perform a runtime test early in your application execution, then use `poll` only if that test succeeds. By testing for support at runtime, your application can use the `poll` API if is supported by a particular version of any operating system, falling back gracefully if it is not supported.

A good rule is to always test for the most specific thing that is guaranteed to meet your requirements. If you need a framework, test for the framework. If you need a library, test for the library. If you need a particular compiler version, test the compiler version. And so on. By doing this, you increase your chances that your application will continue to work correctly without modification in the future.

Choosing a Compiler

OS X ships two compilers and their corresponding toolchains. The default compiler is based on GCC 4.2. In addition, a compiler based on GCC 4.0 is provided. Older versions of Xcode also provide prior versions. Compiling for 64-bit PowerPC and Intel-based Macintosh computers is only supported in version 4.0 and later. Compiling 64-bit kernel extensions is only supported in version 4.2 and later.

Always try to compile your software using GCC 4 because future toolchains will be based on GCC version 4 or later. However, because GCC 4 is a relatively new toolchain, you may find bugs that prevent compiling certain programs.

Use of the legacy GCC 2.95.2-based toolchain is strongly discouraged unless you have to maintain compatibility with OS X version 10.1.

If you run into a problem that looks like a compiler bug, try using a different version of GCC. You can run a different version by setting the `CC` environment variable in your Makefile. For example, `CC=gcc-4.0` chooses GCC 4.0. In Xcode, you can change the compiler setting on a per-project basis or a per-file basis by selecting a different compiler version in the appropriate build settings inspector.

Setting Compiler Flags

When building your projects in OS X, simply supplying or modifying the compiler flags of a few key options is all you need to do to port most programs. These are usually specified by either the `CFLAGS` or `LDFLAGS` variable in your makefile, depending on which part of the compiler chain interprets the flags. Unless otherwise specified, you should add these flags to `CFLAGS` if needed.

Note: The 64-bit toolchain in OS X v10.4 and later has additional compiler flags (and a few deprecated flags). These are described in more detail in *64-Bit Transition Guide*.

Some common flags include:

`-flat_namespace` (in `LDFLAGS`)
: Changes from a two-level to a single-level (flat) namespace. By default, OS X builds libraries and applications with a two-level namespace where references to dynamic libraries are resolved to a definition in a specific dynamic library when the image is built. Use of this flag is generally discouraged, but in some cases, is unavoidable. For more information, see "Understanding Two-Level Namespaces" (page 28).

`-bundle` (in `LDFLAGS`)
: Produces a Mach-O bundle format file, which is used for creating loadable plug-ins. See the `ld` man page for more discussion of this flag.

`-bundle_loader` *executable* (in `LDFLAGS`)
: Specifies which executable will load a plug-in. Undefined symbols in that bundle are checked against the specified executable as if it were another dynamic library, thus ensuring that the bundle will actually be loadable without missing symbols.

`-framework` *framework* (in `LDFLAGS`)

Links the executable being built against the listed framework. For example, you might add -framework vecLib to include support for vector math.

`-mmacosx-version-min` *version*

Specifies the version of OS X you are targeting. You must target your compile for the *oldest* version of OS X on which you want to run the executable. In addition, you should install and use the cross-development SDK for that version of OS X. For more information, see *SDK Compatibility Guide*.

Note: OS X also uses this value to determine the UNIX conformance behavior of some APIs. For more information, read *Unix 03 Conformance Release Notes*.

Note: OS X uses a single-pass linker. Make sure that you put your framework and library options after the object(`.o`) files. To get more information about the Apple linker read the manual page for `ld`.

More extensive discussion for the compiler in general can be found at http://developer.apple.com/releasenotes/DeveloperTools/.

Understanding Two-Level Namespaces

By default, OS X builds libraries and applications with a two-level namespace. In a two-level namespace environment, when you compile a new dynamic library, any references that the library might make to other dynamic libraries are resolved to a definition in those *specific* dynamic libraries.

The two-level namespace design has many advantages for Carbon applications. However, it can cause problems for many traditional UNIX applications if they were designed to work in a flat namespace environment.

For example, suppose one library, call it `libfoo`, uses another library, `libbar`, for its implementation of the function `barIt`. Now suppose an application wants to override the use of `libbar` with a compressed version, called `libzbar`. Since `libfoo` was linked against `libbar` at compile time, this is not possible without recompiling `libfoo`.

To allow the application to override references made by `libfoo` to `libbar`, you would use the flag `-flat_namespace`. The `ld` man page has a more detailed discussion of this flag.

If you are writing libraries from scratch, it may be worth considering the two-level namespace issue in your design. If you expect that someone may want to override your library's use of another library, you might have an initializer routine that takes pointers to the second library as its arguments, and then use those pointers for the calls instead of calling the second library directly.

Alternately, you might use a plug-in architecture in which the calls to the outside library are made from a plug-in that could be easily replaced with a different plug-in for a different outside library. See "Dynamic Libraries and Plug-ins" (page 29) for more information.

For the most part, however, unless you are designing a library from scratch, it is not practical to avoid using `-flat_namespace` if you need to override a library's references to another library.

If you are compiling an executable (as opposed to a library), you can also use `-force_flat_namespace` to tell `dyld` to use a flat namespace when loading any dynamic libraries and bundles loaded by the binary. This is usually not necessary, however.

Executable Format

The only executable format that the OS X kernel understands is the Mach-O format. Some bridging tools are provided for classic Macintosh executable formats, but Mach-O is the native format. It is very different from the commonly used Executable and Linking Format (ELF). For more information on Mach-O, see *OS X ABI Mach-O File Format Reference*.

Dynamic Libraries and Plug-ins

Dynamic libraries and plug-ins behave differently in OS X than in other operating systems. This section explains some of those differences.

Using Dynamic Libraries at Link Time

When linking an executable, OS X treats dynamic libraries just like libraries in any other UNIX-based or UNIX-like operating system. If you have a library called `libmytool.a`, `libmytool.dylib`, or `libmytool.so`, for example, all you have to do is this:

```
ld a.o b.o c.o ... -L/path/to/lib -lmytool
```

As a general rule, you should avoid creating static libraries (`.a`) except as a temporary side product of building an application. You *must* run `ranlib` on any archive file before you attempt to link against it.

Using Dynamic Libraries Programmatically

OS X makes heavy use of dynamically linked code. Unlike other binary formats such as ELF and xcoff, Mach-O treats plug-ins differently than it treats shared libraries.

The preferred mechanism for dynamic loading of shared code, beginning in OS X v10.4 and later, is the `dlopen` family of functions. These are described in the man page for `dlopen`. The `ld` and `dyld` man pages give more specific details of the dynamic linker's implementation.

> **Note:** By default, the names of dynamic libraries in OS X end in `.dylib` instead of `.so`. You should be aware of this when writing code to load shared code in OS X.

Libraries that you are familiar with from other UNIX-based systems might not be in the same location in OS X. This is because OS X has a single dynamically loadable framework, `libSystem`, that contains much of the core system functionality. This single module provides the standard C runtime environment, input/output routines, math libraries, and most of the normal functionality required by command-line applications and network services.

The `libSystem` library also includes functions that you would normally expect to find in libc and libm, RPC services, and a name resolver. Because `libSystem` is automatically linked into your application, you do not need to explicitly add it to the compiler's link line. For your convenience, many of these libraries exist as symbolic links to `libSystem`, so while explicitly linking against `-lm` (for example) is not needed, it will not cause an error.

To learn more about how to use dynamic libraries, see *Dynamic Library Programming Topics* .

Compiling Dynamic Libraries and Plugins

For the most part, you can treat dynamic libraries and plugins the same way as on any other platform if you use GNU libtool. This tool is installed in OS X as `glibtool` to avoid a name conflict with NeXT libtool. For more information, see the manual page for `glibtool`.

You can also create dynamic libraries and plugins manually if desired. As mentioned in "Using Dynamic Libraries Programmatically" (page 30), dynamic libraries and plugins are not the same thing in OS X. Thus, you must pass different flags when you create them.

To create dynamic libraries in OS X, pass the `-dynamiclib` flag.

To create plugins, pass the `-bundle` flag.

Because plugins can be tailored to a particular application, the OS X compiler provides the ability to check these plugins for loadability at compile time. To take advantage of this feature, use the `-bundle_loader` flag. For example:

```
gcc -bundle a.o b.o c.o -o mybundle.bundle \
    -bundle_loader /Applications/MyApp.app/Contents/MacOS/MyApp
```

If the compiler finds symbol requests in the plugin that cannot be resolved in the application, you will get a link error. This means that you `must` use the `-l` flag to link against any libraries that the plugin requires as though you were building a complete executable.

Important: OS X does not support the concept of weak linking as it is found in systems like Linux. If you override one symbol, you must override all of the symbols in that object file.

To learn more about how to create and use dynamic libraries, see *Dynamic Library Programming Topics* .

Bundles

In the OS X file system, some directories store executable code and the software resources related to that code in one discrete package. These packages, known as bundles, come in two varieties: application bundles and frameworks.

There are two basic types of bundles that you should be familiar with during the basic porting process: application bundles and frameworks. In particular, you should be aware of how to use frameworks, since you may need to link against the contents of a framework when porting your application.

Application Bundles

Application bundles are special directories that appear in the Finder as a single entity. Having only one item allows a user to double-click it to get the application with all of its supporting resources. If you are building Mac apps, you should make application bundles. Xcode builds them by default if you select one of the application project types. More information on application bundles is available in "Bundles vs. Installers" (page 64) and in *Mac Technology Overview* .

Frameworks

A framework is a type of bundle that packages a shared library with the resources that the library requires. Depending on the library, this bundle could include header files, images, and reference documentation. If you are trying to maintain cross-platform compatibility, you may not want to create your own frameworks, but you should be aware of them because you might need to link against them. For example, you might want to link against the Core Foundation framework. Since a framework is just one form of a bundle, you can do this by linking against `/System/Library/Frameworks/CoreFoundation.framework` with the `-framework` flag. A more thorough discussion of frameworks is in *Mac Technology Overview*.

For More Information

You can find additional information about bundles in *Mac Technology Overview*.

Handling Multiply Defined Symbols

A multiply defined symbol error occurs if there are multiple definitions for any symbol in the object files that you are linking together. You can specify the following flags to modify the handling of multiply defined symbols under certain circumstances:

`-multiply_defined treatment`
: Specifies how multiply defined symbols in dynamic libraries should be treated when `-twolevel_namespace` is in effect. The values for treatment must be one of:

  `error`—Treat multiply defined symbols as an error.

  `warning`—Treat multiply defined symbols as a warning.

  `suppress`—Ignore multiply defined symbols.

  The default behavior is to treat multiply defined symbols in dynamic libraries as warnings when `-twolevel_namespace` is in effect.

`-multiply_defined_unused treatment`
: Specifies how unused multiply defined symbols in dynamic libraries should be treated when `-twolevel_namespace` is in effect. An unused multiply defined symbol is a symbol defined in the output that is also defined in one of the dynamic libraries, but in which but the symbol in the dynamic library is not used by any reference in the output. The values for treatment must be `error`, `warning`, or `suppress`. The default for unused multiply defined symbols is to suppress these messages.

Predefined Macros

The following macros are predefined in OS X:

`__OBJC__`
: This macro is defined when your code is being compiled by the Objective-C compiler. By default, this occurs when compiling a `.m` file or any header included by a `.m` file. You can force the Objective-C compiler to be used for a `.c` or `.h` file by passing the `-ObjC` or `-ObjC++` flags.

`__cplusplus`
: This macro is defined when your code is being compiled by the C++ compiler (either explicitly or by passing the `-ObjC++` flag).

`__ASSEMBLER__`
: This macro is defined when compiling `.s` files.

`__NATURAL_ALIGNMENT__`
: This macro is defined on systems that use natural alignment. When using natural alignment, an `int` is aligned on `sizeof(int)` boundary, a short int is aligned on `sizeof(short)` boundary, and so on. It is defined by default when you're compiling code for PowerPC architecutres. It is not defined when you use the `-malign-mac68k` compiler switch, nor is it defined on Intel architectures.

`__MACH__`
: This macro is defined if Mach system calls are supported.

`__APPLE__`
: This macro is defined in any Apple computer.

`__APPLE_CC__`
: This macro is set to an integer that represents the version number of the compiler. This lets you distinguish, for example, between compilers based on the same version of GCC, but with different bug fixes or features. Larger values denote later compilers.

`__BIG_ENDIAN__` and `__LITTLE_ENDIAN__`
: These macros tell whether the current architecture uses little endian or big endian byte ordering. For more information, see "Compiling for Multiple CPU Architectures" (page 20).

Note: To define a section of code to be compiled on OS X system, you should define a section using `__APPLE__` with `__MACH__` macros. The macro `__UNIX__` is not defined in OS X.

Other Porting Tips

This section describes alternatives to certain commonly used APIs.

Headers

The following headers commonly found in UNIX, BSD, or Linux operating systems are either unavailable or are unsupported in OS X:

`alloc.h`

This file does not exist in OS X, but the functionality does exist. You should include `stdlib.h` instead. Alternatively, you can define the prototypes yourself as follows:

```
#ifndef _ALLOCA_H

#undef  __alloca

/* Now define the internal interfaces.  */
extern void *__alloca (size_t __size);

#ifdef  __GNUC__
# define __alloca(size) __builtin_alloca (size)
#endif /* GCC.  */

#endif
```

`ftw.h`

The `ftw` function traverses through the directory hierarchy and calls a function to get information about each file. However, there isn't a function similar to `ftw` in `fts.h`.

One alternative is to use `fts_open`, `fts_children`, and `fts_close` to implement such a file traversal. To do this, use the `fts_open` function to get a handle to the file hierarchy, use `fts_read` to get information on each file, and use `fts_children` to get a link to a list of structures containing information about files in a directory.

Alternatively, you can use `opendir`, `readdir` and `closedir` with recursion to achieve the same result.

For example, in order to get a description of each file located in `/usr/include` using `fts.h`, then the code would be as follows:

```
/* assume that the path "/usr/include" has been passed through argv[3]*/

fileStruct = fts_open(&argv[3], FTS_COMFOLLOW, 0);
```

```
dirList = fts_children(fileStruct, FTS_NAMEONLY);

do
{
 fileInfo = fts_read(dirList->fts_pointer);

 /* at this point, you would be able to extract information from the
 FTSENT returned by fts_read */

 fileStruct = fts_open(dirList->fts_link->fts_name,
FTS_PHYSICAL, (void *)result);

}while (dirList->fts_link != NULL);

ftsResult = fts_close(fileStruct);
```

See the manual page for `fts` to understand the structures and macros used in the code. The sample code above shows a very simplistic file traversal. For instance, it does not consider possible subdirectories existing in the directory being searched.

`getopt.h`
Not suported, use `unistd.h` instead.

`lcrypt.h`
Not supported, use `unistd.h` instead.

`malloc.h`
Not supported, use `stdlib.h` instead.

`mm.h`
This header is supported in Linux for memory mapping, but is not supported in Max OS X. In OS X, you can use `mmap` to map files into memory. If you wish to map devices, use the I/O Kit framework instead.

`module.h`
Kernel modules should be loaded using the KextManager API in the I/O Kit framework. For more information, see `KextManagerLoadKextWithURL` and related functions. The modules themselves must be compiled against the Kernel framework. For more information, see *I/O Kit Fundamentals* .

`nl_types.h`
Use the `CFBundleCopyLocalizedString` API in Core Foundation for similar localization functionality.

`ptms.h`
: Although pseudo-TTYs are supported in OS X, this header is not. The implementation of pseudo-TTYs is very different from Linux. For more information, see the `pty` manual page.

`stream.h`
: This header file is not present in OS X. For file streaming, use `iostream.h`.

`stropts.h`
: Not supported.

`swapctl.h`
: OS X does not support this header file. You can use the header file `swap.h` to implement swap functionality. The `swap.h` header file contains many functions that can be used for swap tuning.

`termio.h`
: This header file is obsolete, and has been replaced by `termios.h`, which is part of the POSIX standard. These two header files are very similar. However, the `termios.h` does not include the same header files as `termio.h`. Thus, you should be sure to look directly at the `termios.h` header to make sure it includes everything your application needs.

`utmp.h`
: Deprecated, use `utmpx.h` instead.

`values.h`
: Not supported, use `limits.h` instead.

`wchar.h`
: Although this functionality is available, you should generally use the `CFStringRef` API in Core Foundation instead.

Functions

The following functions commonly seen in other Unix, Linux, or BSD operating systems are not supported or are discouraged in OS X.

`btowc, wctob`
: Although OS X supports the `wchar` API, the preferred way to work with international strings is the `CFStringRef` API, which is part of the Core Foundation framework. Some of the APIs available in Core Foundation are `CFStringGetSystemEncoding`, `CFStringCreateCopy`, `CFStringCreateMutable`, and so on. See the Core Foundation documentation for a complete list of supported APIs.

`catopen, catgets, catclose`

`nl_types.h` is not supported, thus, these functions are not supported. These functions gives access to localized strings. There is no direct equivalent in OS X. In general, you should use Core Foundation to access localized resources (`CFBundleCopyLocalizedString`, for example).

`crypt`

The `crypt` function performs password encryption, based on the NBS Data Encryption Standard (DES). Additional code has been added to deter key search attempts.`crypt`. OS X's version of the function `crypt` behaves very similarly to the Linux version, except that it encrypts a 64-bit constant rather than a 128-bit constant. This function is located in the `unistd.h` header file rather than `lcrypt.h`.

Note: The linker flag `-lcrypt` is not supported in OS X.

`dysize`

This function is not supported in OS X. Its calculation for leap year is based on:

```
(year) %4 == 0 && ((year) %100 != 0 || (year) % 400 == 0)
```

You can either use this code to implement this functionality, or you can use any of the existing APIs in time.h to do something similar.

`ecvt`, `fcvt`

Discouraged in OS X. Use `sprintf`, `snprintf`, and similar functions instead.

`fcloseall`

This function is an extension to `fclose`. Although OS X supports `fclose`, `fcloseall` is not supported. You can use `fclose` to implement `fcloseall` by storing the file pointers in an array and iterating through the array.

`getmntent`, `setmntent`, `addmntent`, `endmntent`, `hasmntopt`

In general, volumes in OS X are not in `/etc/fstab`. However, to the extent that they are, you can get similar functionality from `getfsent` and related functions.

`poll`

This API is partially supported in OS X. It does not support polling devices.

`sbrk`, `brk`

The `brk` and `sbrk` functions are historical curiosities left over from earlier days before the advent of virtual memory management. Although they are present on the system, they are not recommended.

`shmget`

This API is supported but is not recommended. `shmget` has a limited memory blocks allocation. When several applications use `shmget`, this limit may change and cause problems for the other applications.

In general, you should either use `mmap` for mapping files into memory or use the POSIX `shm_open` function and related functions for creating non-file-backed shared memory.

`swapon`, `swapoff`

These functions are not supported in OS X.

Utilities

The chapter "Designing Scripts for Cross-Platform Deployment" in *Shell Scripting Primer* describes a number of cross-platform compatibility issues that you may run into with command-line utilities that are commonly scripted.

This section lists several commands that are primarily of interest to people porting compiled applications and drivers, rather than general-purpose scripting.

`ldd`

The `ldd` command is not available in OS X. However, you can use the command `otool -L` to get the same functionality that `ldd` provides. The `otool` command displays specified parts of object files or libraries. The option `-L` displays the name and version numbers of the shared libraries that an object file uses. To see all the existing options, see the manual page for `otool`.

`lsmod`

`lsmod` is not available on OS X, but other commands exist that offer similar functionality. The

`kextutil`

Loads, diagnoses problems with, and generates symbols for kernel extensions.

`kextstat`

Prints statistics about currently loaded drivers and other kernel extensions.

`kextload`

Loads the kernel module for a device driver or other kernel extensions. This command is a basic command intended for use in scripts. For developer purposes, use `kextutil` instead.

`kmodunload`

Unloads the kernel module for a device driver or other kernel extensions. This command is a basic command intended for use in scripts. For developer purposes, use `kextutil` instead.

For more information about loading kernel modules, see *Kernel Extension Programming Topics* .

Using OS X Native Application Environments

OS X includes three high-level native development environments that you can use for your application's graphical user interface: Carbon, Cocoa, and Java. These environments are full-featured development environments in their own right, and you can write complete applications in any one of these environments.

In addition to these technologies, you can also use OpenGL, X11, Qt, Tcl/Tk, wxWidgets, and a number of other traditional UNIX graphics technologies when writing applications for OS X. In each case, there are tradeoffs. This chapter explains these tradeoffs and shows how they affect you as a programmer as well as your users.

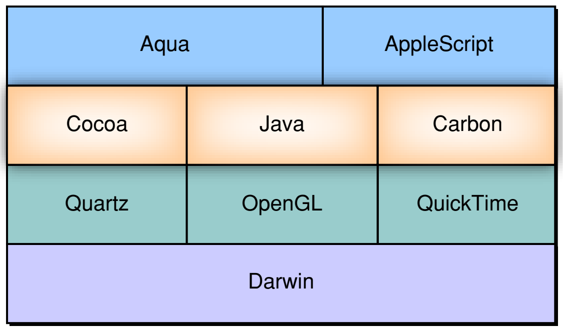
Choosing a Native Application Environment

In the context of this document, Carbon, Cocoa, and Java are presented as a way to write a GUI wrapper for a UNIX-based tool.

Writing to these environments lets you build an application that is indistinguishable from any other native Mac app. The Java or Cocoa frameworks are probably the environments that you will use in bringing UNIX applications to OS X, although the Carbon frameworks are used by some developers. The following sections outline all three.

A good general rule is that if you decide to port your application from X11 to a native GUI environment, Carbon is a good choice for C and other procedural X11 applications, while Cocoa is a good choice if your application uses high-level toolkits such as Tcl/Tk or Qt. Cocoa is also a good choice for wrapping command-line applications with a GUI.

Figure 4-1 High-level graphics technologies



If you decide to port your cross-platform application to a native OS X GUI environment, you should carefully consider how best to do this port in a way that is maintainable. For some helpful tips, you should also read "(Re)designing for Portability" (page 78).

Cocoa

Cocoa is an object-oriented framework that incorporates many of OS X's greatest strengths. Based on the highly-respected OpenStep frameworks, it allows for rapid development and deployment with both its object-oriented design and integration with the OS X development tools. Cocoa is divided into two major parts: Foundation and the Application Kit. Foundation provides the fundamental classes that define data types and collections; it also provides classes to access basic system information like dates and communication ports. The Application Kit builds on that by giving you the classes you need to implement graphical event-driven user interfaces.

Cocoa also provides file system abstraction that makes things such as file browsers look like Mac users expect. If you are developing a commercial application, you should use these. If you are developing an in-house application that will largely be used by people familiar with UNIX, the layout may be confusing, so these APIs may not be appropriate. See "Presenting File Open and Save Dialog Boxes" (page 61) for more information.

The native language for Cocoa is Objective-C, which provides object-oriented extensions to standard C and Objective-C++. *The Objective-C Programming Language* describes the grammar of Objective-C and presents the concepts behind it. Objective-C is supported in gcc 2.95 and 3. Most of the Cocoa API is also accessible through Java. Cocoa is also the basis for AppleScript Studio, an application environment for script-based GUI development.

Additional Cocoa information, including sample code, can be found at http://developer.apple.com/. Cocoa documentation including tutorials is available at http://developer.apple.com/.

Benefits of Cocoa Development

- Rapid development environment
- Object-oriented framework design
- Excellent integration with OS X developer tools
- Very robust feature set
- Can take advantage of existing C, C++, Objective-C, and Java code
- Similar level of abstraction to high level toolkits such as Qt, Tcl/Tk, and so on

Drawbacks of Cocoa Development

- Cross-platform deployment requires having a separate, non-Cocoa code base for the GUI portion

- Requires integrating Java, Objective C, or Objective C++ into your code base for the GUI layer
- Requires knowledge of Java, Objective C, or Objective C++
- Very different level of abstraction from raw X11 programming
- Potential performance penalties if used incorrectly

Example: Calling C or C++ Code With Cocoa

When designing an application from the ground up in Cocoa, you are in an object-oriented environment. You can also take advantage of Cocoa's object-oriented nature when converting preexisting code, as you can use the Cocoa frameworks to wrap the functionality of C or C++ code.

The Objective-C language has been extended to understand C++. Often this is called Objective-C++, but the functionality remains basically the same. Because Cocoa understands Objective-C++, you can call native C and C++ code from Cocoa. This is one example of how you can take advantage of your code base while adding a Macintosh front end. An example is provided below of how Objective-C wraps together C, C++, and Objective-C++ functionality. Listing 4-1 (page 41) shows the Objective-C main class.

Listing 4-1 `main.m`

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


int main(int argc, const char *argv[]) {
    return NSApplicationMain(argc, argv);
}
```

This gets everything started when the user double-clicks the application icon. A call is then sent to invoke a `HelloController` object by the NIB, a file that holds interface information. The listings for `HelloController.m` and `HelloController.h` follow.

Listing 4-2 `HelloController.m`

```
#import "HelloController.h"


@implementation HelloController


- (void)doAbout:(id)sender
{
    NSRunAlertPanel(@"About",@"Welcome to Hello World!",@"OK",NULL,NULL);
}

- (IBAction)switchMessage:(id)sender
{
        int which=[sender selectedRow]+1;
        [helloButton setAction:NSSelectorFromString([NSString
stringWithFormat:@"%@%d:",@"message",which])];
}

- (void)awakeFromNib
{
    [[helloButton window] makeKeyAndOrderFront:self];
}

@end
```

Listing 4-3 `HelloController.h`

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

@interface HelloController : NSObject
{
    id helloButton;
    id messageRadio;
}
- (void)doAbout:(id)sender;
- (void)switchMessage:(id)sender;
@end
```

The communication between the C, C++, and the Objective-C code is handled as shown in Listing 4-4 (page 42). The header file `SayHello.h` is shown in Listing 4-5 (page 43).

Listing 4-4 `SayHello.mm`

```
#import "SayHello.h"
```

```
#include "FooClass.h"
#include <Carbon/Carbon.h>

@implementation SayHello

- (void)message1:(id)sender
{
    NSRunAlertPanel(@"Regular Obj-C from Obj-C",@"Hello, World! This is a regular
 old NSRunAlertPanel call in Cocoa!",@"OK",NULL,NULL);
}

- (void)message2:(id)sender
{
    int howMany;
    NSString *theAnswer;
    Foo* myCPlusPlusObj;
    myCPlusPlusObj=new Foo();
    howMany=myCPlusPlusObj->getVariable();
    delete myCPlusPlusObj;
    theAnswer=[NSString stringWithFormat:@"Hello, World! When our C++ object is
queried, it tells us that the number is %i!",howMany];
    NSRunAlertPanel(@"C++ from Obj-C",theAnswer,@"OK",NULL,NULL);
}

- (void)message3:(id)sender
{
    Alert(128,NULL); //This calls into Carbon
}

@end
```

Listing 4-5 `SayHello.h`

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

```
@interface SayHello : NSObject
{
}

- (void)message1:(id)sender;
- (void)message2:(id)sender;
- (void)message3:(id)sender;
@end
```

The C++ class wrapped by these Cocoa calls is shown in Listing 4-6 (page 44). The header file, `FooClass.h`, is shown in Listing 4-7 (page 44).

Listing 4-6 `FooClass.cpp`

```
#include "FooClass.h"
Foo::Foo()
{
 variable=3;
}

int Foo::getVariable()
{
 return variable;
}
```

Listing 4-7 `FooClass.h`

```
class Foo {
public:
    Foo();
    int getVariable();
    void * objCObject;
private:
    int variable;
};
```

You should be careful when writing code using Cocoa, because the same constructs that make it easy for the developer tend to degrade performance when overused. In particular, heavy use of message passing can result in a sluggish application.

The ideal use of Cocoa is as a thin layer on top of an existing application. Such a design gives you not only good performance and ease of GUI design, but also minimizes the amount of divergence between your UNIX code base and your OS X code base.

Carbon

Carbon is an environment designed to bring existing Mac OS applications to OS X. It can be used to bring an X11 application to a native OS X environment, since the basic drawing primitives are at a similar level of abstraction.

Carbon also offers relatively straightforward file I/O with a few enhancements that are unavailable through POSIX APIs, such as aliases (which behave like a cross between a symbolic link and a hard link). Carbon presents the file system differently to the user than UNIX applications do. This is the layout that Mac users are used to seeing, and thus you should use it if possible when developing a commercial application for broad-scale deployment.

However, if your users are predominantly UNIX users (for example, if you are an in-house developer in a corporate environment), the use of the Carbon file API may not be appropriate, since volumes appear as they do in the Finder, rather than being organized according to the file system hierarchy.

Benefits of Carbon Development

- Well-documented feature set
- Integration with OS X developer tools
- Very robust feature set
- Simple C and C++ integration
- Similar abstraction level to X11
- Procedural design rather than object-oriented

Drawbacks of Carbon Development

- No cross-platform deployment without a separate non-Carbon code base
- Slightly more effort required to take advantage of native OS X technologies
- Procedural design rather than object-oriented

In case you're wondering, that isn't a typo. Procedural design in a GUI can be both a benefit and a drawback.

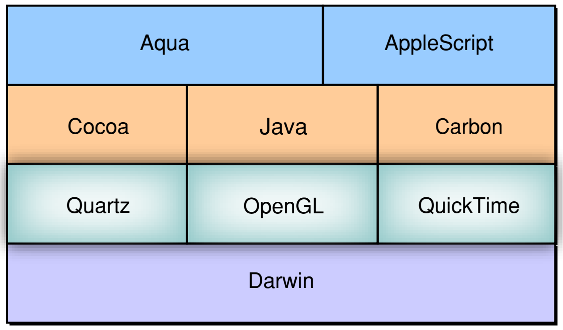
Procedural design can be a benefit in terms of being able to easily integrate into any environment, regardless of language. It is also a benefit because it fits many styles of graphical programming well.

Procedural design can be a drawback, however, when dealing with more complex GUI designs where it would be more convenient to have the methods more closely tied to the data on which they operate.

Lower-Level Graphics Technologies

With some code it is simple to abstract the display code from the underlying computational engine, but often this isn't the case. Especially for graphics-intensive applications, you may want to take advantage of directly calling the core graphic functionality of your targeted operating system. You still need to wrap this functionality in a higher-level API to display it to the screen and allow user interaction, but for times where you need to push pixels in very specific ways, OS X offers you access to three first-class graphics technologies: Quartz, OpenGL, and QuickTime.

Figure 4-2 Low-level graphics technologies



Quartz

Quartz is the graphical system that forms the foundation of the imaging model for OS X. Quartz gives you a standards-based drawing model and output format. Quartz provides both a two-dimensional drawing engine and the OS X windowing environment. Its drawing engine leverages the Portable Document Format (PDF) drawing model to provide professional-strength drawing. The windowing services provide low-level functionality such as window buffering and event handling as well as translucency. Quartz is covered in more detail in *Mac Technology Overview* and at (http://developer.apple.com/).

Benefits of using Quartz

- PostScript-like drawing features
- PDF-based
- Included color management tools

- Unified print and imaging model

Drawbacks to using Quartz

- No cross-platform deployment

OpenGL

OpenGL is an industry-standard 2D and 3D graphics technology. It provides functionality for rendering, texture mapping, special effects, and other visualization functions. It is a fundamental part of OS X and is implemented on many other platforms.

Given its integration into many modern graphics chipsets and video cards, it is of special interest for programs that require intricate graphic manipulation. OpenGL's homepage gives more information on the technology in general at http://www.opengl.org.

Apple's OpenGL page, at https://developer.apple.com/devcenter/mac/resources/opengl/, gives more information on how it is integrated into OS X and explains the enhancements that the OS X OpenGL implementation provides.

Benefits of using OpenGL

- Cross-platform technology
- Native OS X integration
- Included with every installation of OS X
- Very robust feature set for handling graphics

Drawbacks to using OpenGL

- Some level of integration at an application level is required

QuickTime

QuickTime is a powerful multimedia technology for manipulating, enhancing, storing, and delivering graphics, video, and sound. It is a cross-platform technology that provides delivery on Mac OS as well as Windows. More information on the technology can be found at http://developer.apple.com/quicktime/.

Benefits of using QuickTime

- Robust feature set for manipulating sound and video
- Cross-platform development

Drawbacks to using QuickTime

- Not supported on other UNIX-based systems.

Note that while QuickTime itself is not supported on UNIX-based systems, the QuickTime file format is supported by various third-party utilities.

Choosing a Graphical Environment for Your Application

OS X offers many options for transforming your applications with a graphical user interface from a UNIX code base to a native OS X code base, or even for wrapping preexisting command-line tools or utilities with a graphical front end, making them available to users who never want to go to the command line.

This chapter describes some of the issues you will face when porting a GUI application to OS X or adding a GUI wrapper around a command line application. It also describes the various GUI environments supported by OS X and gives advantages and disadvantages of each.

In choosing a graphical environment to use in bringing a UNIX-based application to OS X, you will need to answer the questions posed in the following sections:

- "What Kind of Application Are You Porting?" (page 50)
- "How Well Does It Need to Integrate With OS X?" (page 50)
- "Does Your Application Require Cross-Platform Functionality?" (page 51)

These questions should all be evaluated as you weigh the costs and benefits of each environment. You may already be using a cross-platform toolkit API. If you aren't doing so, you may want to port your application to a native API such as Carbon or Cocoa.

If you are a commercial developer adding a new graphical user interface to a command-line application and want to take advantage of the greatest strengths of OS X, you will probably want to use the Cocoa API. In some cases, you may want to use a different API for reasons such as cross-platform compatibility.

If you decide to use a nonnative API, like X11R6, to provide a user interface for your Mac app, it is important to remember that users and developers with a UNIX background might be perfectly content to just have the application running in OS X. Traditional Macintosh users, however, will pass up an application with a traditional UNIX interface for a more integrated and modern interface. Whether you target a straight port to the Darwin layer or a more robust transformation of your application to take advantage of other OS X technologies (like the Cocoa frameworks) is your decision.

What Kind of Application Are You Porting?

Are you bringing a preexisting code base to OS X, or are you adding new functionality—for example a graphical interface—to a command-line application? If you already have a code base written to a particular API, and that API is supported in OS X, you probably want to continue using that API for any large, complex application unless you desire features of another API.

For simple applications, or for applications where you are wrapping a command-line utility with a graphical user interface, you need to evaluate what API to use. Reading the next two sections will help you recognize the benefits and drawbacks of each technology.

How Well Does It Need to Integrate With OS X?

Who are you marketing your application to? If they are traditional UNIX users that just want to run, for example, a gene-sequencing application alongside Microsoft Office, then it may be sufficient to just install an X Window System on their OS X computer. You would simply port your X11R6-based application to OS X, leaving your code as it stands (aside from the little changes you may need to make to make it compile in OS X). For more information on X11 in OS X, see "X11R6" (page 54).

If you are an in-house developer of UNIX applications, this may be as far as you want to go, particularly if you want to maintain the same user experience across multiple platforms. However, you may still want to use Carbon or Cocoa APIs to improve the overall look of the UI to make it easier to use.

If you sell an application, some customers might at first be happy just to have it on their platform. However, if a competing product is released using OS X native functionality, customers are likely to gravitate to that product.

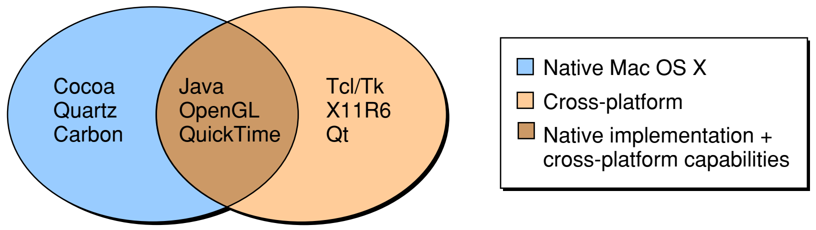
A hot topic in the science and technology industries is not only bringing a code base to OS X, but also giving that application an OS X native user interface. This is not a decision to be made trivially for an application with a complex GUI and a large code base, but it is one that can make or break a product's success in the market. The individual discussions of the APIs that follow should help you to make a well-informed decision.

If you're an open source software developer, you will probably gravitate toward a basic port of the X11 application. However, you should consider creating a native GUI if your application is likely to be used by consumers such as a word processor or a web browser.

Does Your Application Require Cross-Platform Functionality?

If you have an application that must work on multiple platforms, OS X has you set up for success. You have many options; some are built in and shipped with every version of the operating system; others require the installation of additional components. Figure 5-1 (page 51) depicts the distinction between the cross-platform APIs that are native and those that aren't.

Figure 5-1 Graphical environments



You can see that OS X includes some standard cross-platform APIs: Java, OpenGL, and QuickTime. There are also commercial and free implementations of some of the traditional UNIX technologies. If you are building a cross-platform application, you should evaluate which platforms you are targeting with your application and determine which API allows you to bring your UNIX-based application to OS X. Table 5-1 (page 51) lists the platforms on which OS X cross-platform technologies run.

Table 5-1 Platforms of cross-platform technologies

API	Platforms
"OpenGL" (page 47)	OS X, UNIX-based systems, Windows
"QuickTime" (page 47)	OS X, Windows
"Qt" (page 55)	OS X (Native & X11), UNIX-based systems, Windows
"Tcl/Tk" (page 55)	OS X (Native & X11), UNIX-based systems, Windows
"wxWidgets" (page 56)	OS X (Native & X11), UNIX-based systems, Windows
"X11R6" (page 54)	OS X, UNIX-based systems

Note: Although some of the technologies in Table 5-1 (page 51) are supported in Mac OS 9, support for these technologies in Mac OS 9 is not considered here because the rest of your UNIX code base will not run in Mac OS 9.

In the next two chapters, you'll find brief descriptions of each of the technologies available to you for your application's graphical user interface.

Using Traditional UNIX Graphical Environments

UNIX-based operating systems have grown to include many environments for providing a graphical interface to users. The X Window System, also known as X11, is probably the most well known. As more specific needs have arisen, other architectures have been developed that assume an X11 implementation.

OS X does not use the X Window System by default, but implementations of X11 are available from Apple and from third-party developers. This means that X Window System–based applications can be run, as can many of the alternative UNIX-style graphical environments. This section gives more details on some of these environments.

If your application uses the X Window System you need to either port the user interface to a native OS X environment, provide an X Window System implementation with your application, or require that your users install the Apple X11 package (an installation option in the OS X installer beginning in OS X version 10.3).

If you are a commercial software developer or if your software will be used by traditional end users (including word processors, web browsers, and so on), you may want to consider porting to a native graphical environment instead of requiring X11. However, for internal tools or for tools with a specialized user base, it may be more reasonable to continue using X11. The sections in this chapter will help you decide how to choose an appropriate X11-based environment for your application.

Choosing a UNIX Graphical Environment

Many UNIX applications are designed using high-level widget toolkits such as Tcl/Tk, Motif (or OpenMotif or Lesstif), and Qt. These environments work well across many platforms. However, that cross-platform nature comes with a price in flexibility. In essence, to be cross-platform, they can only support capabilities that are generic to all of the potential operating environments, and as such, they tend to provide only the lowest common denominator in terms of functionality.

Such toolkits are often weak in areas such as performance and flexibility, and many do not allow precise control over how GUI elements are laid out.

If they provide everything you need, then they are good choices. Otherwise, a more flexible alternative such as raw Xlib programming, or a more Mac-tuned alternative such as Carbon or Cocoa may be more appropriate.

Other UNIX solutions exist for more extreme cross-platform environments. One example is MicroWindows, which is designed to allow ease of code sharing between X11 and Windows by implementing a subset of the Win32 graphics APIs on X11. Because this is really just an X11 application, it should be possible to use it on OS X. However, due to the amount of abstraction involved, such a solution will not perform as well as a native application.

If you are trying to design an application for such an environment, it is generally better to rearchitect your code to have multiple front-ends—one for Windows, one for OS X, and one for UNIX (X11). See "(Re)designing for Portability" (page 78) for more information on creating a graphics abstraction layer.

X11 R6

OS X does not include an X11 R6 implementation by default. If you are intent on only porting your application without adding any OS X functionality, you can still run X11 on OS X. Along with your application, you should instruct your users on how to install one.

The easiest way to get an X11 implementation is to install the Apple X11 implementation. Beginning in OS X v10.3, the X11 package is an optional installation that comes on the install DVD or CD.

For *older versions* of OS X (before v10.4), you can download X11 from http://www.apple.com/downloads/macosx/apple/macosx_updates/x11formacosx.html.

Important: Do *not* install the downloadable version of X11 on OS X v10.4. It will not work. The Apple X11 version compatible with v10.4 is available *only* on the OS X install DVD.

In addition, several other commercial and free alternatives exist.

XTools by Tenon Intersystems (http://www.tenon.com) and eXodus from Powerlan USA both provide X11 servers that coexist with Aqua.

If you do not need a commercial implementation of X11, XFree86 offers a very robust free X11 R6 implementation. The OS X version of XFree86 is an active project with integral support from the XQuartz (formerly XDarwin) project. More information on the XQuartz project including tips for installing X11 is available at http://xquartz.macosforge.org/trac/wiki. XFree86 itself can be downloaded from http://xfree86.org/.

With an X Window System implementation, you are now free to use toolkits that you are already familiar with—for example, GTK+ or KDE.

Benefits of X11 R6 Development

- The de facto standard in display technology for UNIX-based operating systems

- Open source implementation is available
- Somewhat portable to certain embedded systems

Drawbacks of X11R6 Development

- Complicated development environment
- Requires substantial disk space for a full installation
- Is not native to OS X. This means your application is treated as a second-class citizen to some extent, particularly in the areas of drag-and-drop and system services.

Tcl/Tk

There is a native Aqua version of Tk available at http://tcl.sourceforge.net. With this you can easily add graphical elements to your existing Perl, Python, or Tcl scripts.

Benefits of Tk Development

- Cross-platform development environment
- Easily integrates with Tcl, Perl, and Python scripts
- Open source implementation is available

Drawbacks of Tk Development

- Not supported by default in OS X
- As with most high-level toolkits, flexibility is limited

Qt

Qt is a C++ toolkit for building graphical user interfaces. It is very popular in the UNIX world, as it is used by the very popular K Desktop Environment, KDE. Trolltech has a native version of Qt, Qt/Mac, available for OS X. If you already have a C++ application or are considering building one, Qt lets you build applications that run natively in OS X as well as Linux and Windows. Information about Qt/Mac is available at http://www.qt.nokia.com. Qt/Mac does not run on top of X11 in OS X, but the source code is compatible with Qt's X11 implementation.

Benefits of Qt Development

- Cross-platform development environment

- Integrates easily with C++ code
- Robust feature set

Drawbacks in Qt Development

- Requires licensing for commercial development
- As with most high-level toolkits, flexibility is limited

GTK+/GTKmm

GTK+ and GTKmm are the C and C++ interfaces (respectively) to another toolkit for building graphical user interfaces. It is popular in the Gimp and Gnome development communities. There is a beta native version of the GTK+ 2.x toolkits available for OS X. If you already have a C or C++ application or are considering building one, GTK+ lets you build applications that run natively in OS X as well as Linux and Windows.

Information about Gtk-MacOSX (an implementation of the GTK+ 2.x toolkit) is available at http://gtk-osx.sourceforge.net/.

These toolkits do not run on top of X11 in OS X, but are source-code–compatible with the equivalent X11 implementations.

Benefits of GTK+/GTKmm Development

- Cross-platform development environment
- Integrates easily with C/C++ code
- Robust feature set

Drawbacks in GTK+/GTKmm Development

- LGPL license reverse engineering clause may raise concerns for some commercial developers
- As with most high-level toolkits, flexibility is limited

wxWidgets

The wxWidgets toolkit is another toolkit for building graphical user interfaces. It is a C++ framework for building native look-and-feel applications in a cross-platform way. It currently supports UNIX/Linux (X11), Windows, and OS X. Support for other platforms is in development.

More information about wxWidgets can be found at http://www.wxwidgets.org.

Wrapping a Command Line Application with a GUI Interface

Command line applications are a good first step in writing software for OS X. However, most OS X users tend to shy away from the command line (former UNIX and Linux users notwithstanding). For this reason, if you want your command line application to have mass appeal, you should consider wrapping it in a GUI interface.

OS X provides a number of methods for wrapping a command line application with a GUI interface. These include:

- `Do Shell Script` (AppleScript Studio)
- `NSTask` (Cocoa)

Each provides different capabilities for different purposes. This section points you to existing documentation on the topic and describes common issues that arise when wrapping a command line application with a GUI interface.

This section is primarily intended for open source developers, shareware/freeware developers, and in-house UNIX application developers, as these techniques do not offer the same level of functionality that you can get through tighter integration between the GUI and the underlying application.

Choosing a Wrapping Method

If you just need to run a shell script by double-clicking it, and have no need for a GUI, you can simply rename the shell script to end with `.command` and it will open in Terminal automatically.

If you need more control over arguments but the command-line tool is non-interactive—that is, the GUI wrapper will merely start it and display the results—then either `NSTask` or `"Do Shell Script"` is acceptable, and you should use whichever one is most comfortable.

If the application needs to interact with the tool in any way, you should use the function `NSTask`. With it, you can chain arbitrary numbers of tasks using the `NSPipe` function.

File Access Considerations

Carbon, Cocoa, AppleScript, and other OS X development environments deal with files by using file references rather than referring to them by their path. As a result, you can move files anywhere on the disk and the file reference will still be valid even though the path has changed.

Using file references presents two problems for the developer. First, you have to explicitly convert them to POSIX paths to use them as the argument to a shell script. Second, the POSIX paths can contain spaces, so you must construct the command line carefully to prevent the script from failing if a directory name (or the name of your hard drive) contains a space.

Because of the potential side effects, you should always use the quoted form of the POSIX path when passing the path to a shell script. This is described in detail in *Technote #2065: Issuing Commands* .

The issue of spaces in pathnames is also a slight issue for Cocoa developers using the `NSTask` API, usually because of bugs in the shell script itself. (You do not need to escape strings passed as arguments using `NSTask`.)

You should always test any GUI-wrapped command-line application with filenames that contain spaces and other strange characters such as double quotes (") and trademark (™) to make sure that there are no unexpected failures.

For More Information

For more information on AppleScript's `Do Shell Script` command, see *Technote #2065: Issuing Commands* , at http://developer.apple.com/technotes/tn2002/tn2065.html or the "simple sheet" example in AppleScript Studio.

For more information on Cocoa's `NSTask` API, see *Interacting with the Operating System* .

Porting File, Device, and Network I/O

OS X offers most of the standard UNIX mechanisms for file and device I/O. There are, however, differences to be aware of when porting your application to OS X from other UNIX-based and UNIX-like platforms.

This chapter describes file I/O and device I/O in OS X, including APIs that will enhance the user experience such as the file manager APIs for file access.

Note: This document does not cover device driver porting. For information on device driver porting, read *Porting Drivers to OS X* .

If you are a commercial software developer or if your application will be used by end users, you should read this chapter.

If you are writing a port of an open source application or an in-house UNIX application, you should read this chapter only if your application already uses or plans to use alternate file APIs for other platforms or if you need to do device I/O.

How OS X File I/O Works

OS X contains all of the standard UNIX and POSIX file I/O functionality. For a basic port of an application to OS X, you generally do not need to make any changes in this area unless your application assumes that the file system stores files in a case-sensitive manner. (HFS+, the default Mac file system, is case-preserving, but not case-sensitive.)

However, traditional Mac applications have some enhanced behavior that you may want to incorporate in your application. The most powerful of these is the use of aliases to find files when their location has changed. It is not possible to use this capability through standard UNIX APIs. To obtain this functionality, you must use either Carbon or Cocoa APIs.

For example, some Mac applications also take advantage of the HFS+ file system's ability to handle multi-forked files. This could be used, for example, to store disposable information about a file such as last window position. It is not recommended that crucial information be stored in resource forks when writing new applications.

However, for compatibility reasons, you may need to access data stored in resource forks if your application needs to read files created by older Mac applications, or if you are writing a backup tool that needs to retain the entire contents of a file.

Also, Mac applications present the file system in a different way than POSIX applications in open and save dialogs. Mac applications have a directory structure multiply rooted from the individual volumes instead of singly rooted from the root volume. Using the Carbon or Cocoa APIs achieves this view of the file system automatically. These APIs also present standard file open and save dialogs that match the standard user experience that Mac users have come to expect.

Before moving to a Carbon or Cocoa file browser interface, you should consider what kinds of users are likely to use your application on the Mac platform. Are they mostly experienced in using another UNIX-based platform, or are they largely Mac users?

If they are mostly UNIX users, the Mac file dialog may be confusing to them. If they are mostly Mac users, a traditional UNIX file dialog may be equally confusing. You should generally choose which file API to use based on long-term expectations rather than the current user base to avoid problems down the line.

Note: Before considering the use of a non-POSIX file API such as those in Carbon or Cocoa, you should read the chapter "(Re)designing for Portability" (page 78) and consider ways to rearchitect your file accesses to minimize the number of platform-conditional pieces of code, both for readability and for debuggability.

Using Carbon APIs for File I/O

Carbon APIs are useful when writing file I/O entirely in C. They provide very basic access much like the traditional POSIX APIs.

Table 7-1 POSIX versus Carbon APIs

POSIX function	Carbon Function	Description
`open`, `fopen`	`FSOpenFork`	Takes a filesystem reference (`FSRef`) as an argument and returns a fork reference number.
`close`, `fclose`	`FSClose`	Takes a fork reference number as an argument.
`create`	`FSCreateFileUnicode`	Takes the `FSRef` of the parent directory and the name of the file to be created.
`mkdir`	`FSCreateDirectory-Unicode`	Takes the `FSRef` of the parent directory and the name of the directory to be created.

POSIX function	Carbon Function	Description
`rmdir`	`FSDeleteObject`	Deletes a file or folder (by `FSRef`).
`unlink`	`FSDeleteObject` or `FSDeleteFork`	Deletes a file or folder (by `FSRef`) or delete a single fork of a file.

Note that these functions use `FSSpec` and `FSRef` structures rather than file names. The functions `FSRefMakePath` and `FSPathMakeRef` convert an `FSRef` to a path and vice versa. Similarly, `FSpMakeFSRef` converts an `FSSpec` to an `FSRef`.

Using Cocoa APIs for File I/O

Cocoa file APIs are very different from traditional POSIX APIs. An explanation of the Cocoa APIs is beyond the scope of this book. For more information on Cocoa file APIs, see *File System Programming Guide* .

Presenting File Open and Save Dialog Boxes

File open and save dialog boxes can be presented in OS X using Carbon or Cocoa file dialog APIs. If you choose to use the standard dialog boxes, use the same API you used for the rest of your GUI.

The Carbon file dialog API, Navigation Services, is described in *Navigation Services Reference* in Carbon Documentation.

The Cocoa file dialog API consists of the functions `NSOpenPanel` and `NSSavePanel`, for open and save dialogs. These are described in *Application File Management* in Cocoa Documentation.

How OS X Device I/O Works

OS X provides many of the traditional UNIX mechanisms for device I/O. However, individual device driver designs determine whether or not to use these mechanisms.

In OS X, disk devices, serial port devices, the random-number generator pseudo-device, and pseudo-tty devices (pttys) are traditionally accessed through standard UNIX-style block and character devices. Other pseudo-devices (devices without hardware backing) can also be implemented as BSD devices. You can use these device files in the same way as you would use them on any other UNIX-based or UNIX-like system.

Most actual hardware devices, however, such as USB and FireWire devices are not handled with block or character devices in OS X. Instead, the primary mechanism for accessing hardware devices is through I/O Kit user clients, which are basically remote procedure call or system call interfaces that allow you to call functions within a kernel driver from an application.

To access a hardware device, you must use APIs to search the device tree for a device matching various parameters. You then call functions within the driver. The mechanism used to call these functions depends on the design of the device interface, as do the functions themselves. You can also get information about the device using a standardized mechanism provided by the I/O Kit to examine its properties in the device tree.

A device cannot be controlled from an application unless there is a driver in the kernel. In many cases, the driver may be a simple pass-through that presents a device interface that allows the real device driver to be part of the application itself. In other cases, it may be a full driver for the device that presents a device interface for configuration. The device interface may be provided by OS X itself (such as the user space device interface for USB HID devices) or it may be provided by the individual hardware vendor (such as an audio card driver).

The implementation of a user client is beyond the scope of this document. It is described in detail in the book *Accessing Hardware From Applications* . Since the design of a user client is also heavily dependent on the design of the device interface to which it is connecting, you may also need to read documentation specific to the technology area in question. In the case of device interfaces created by hardware vendors, you may also need to contact the hardware vendor for additional programming information.

Note: For information on porting actual device drivers, read *Porting Drivers to OS X* .

File System Organization

The OS X file system organization is slightly different from that of most UNIX environments, and is described in detail in the man page `hier`. This section presents only the most important differences between a typical UNIX environment and OS X.

First, OS X has a number of folders intended primarily for use with GUI applications or for parts of OS X itself. These paths generally start with a capital letter, and include:

- `/Applications`—contains GUI applications
- `/System`—contains system frameworks and parts of the OS itself
- `/Library`—contains primarily user-installed frameworks
- `/Developer`—contains the developer tools (if installed)
- `/Users`—contains the home directories of local users

In addition, various ports collections are available, and install files in various locations, such as `/sw` (fink), `/usr/local` (GNU-Darwin), and `/opt/local/` (Darwin Ports).

A few directories, including `/etc` and `/tmp` are actually symbolic links into `/private`. You should be careful not to stomp on these symbolic links.

Finally, you should be aware that by default, a number of directories are hidden from users when viewing the file system using the OS X GUI. These directories include most of the standard system directories, including `/bin`, `/dev`, `/etc`, `/sbin`, `/tmp`, and `/usr`. These directories still appear and behave in the expected manner when accessed from the command line. For more information on making these visible from GUI applications, see "File-System Structure and Visibility" (page 70).

OS X has different privilege sets for file system access. Users by default have write access to their home directory and certain other directories created in previous versions of Mac OS. Admin users have write access to additional parts of the system, including various application directories and configuration files, without the need to become the root user. Finally, the directories containing the OS itself are read-only except as root.

How Mac OS Networking Works

OS X provides the usual POSIX and BSD networking functionality. For more information on the actual APIs, see *OS X Man Pages* .

For the most part, this networking behaves just like it does on any other UNIX system. OS X differs in one crucial way, however. It does not use `/etc/hosts`, `/etc/resolv.conf`, and other similar configuration files for network configuration. (More precisely, the file `/etc/resolv.conf` is provided for read-only use, but should not be modified directly. The file `/etc/hosts` is provided, but is not used by default.)

If you need to configure name server settings (or other network settings), you should use the System Configuration framework (described in *System Configuration Programming Guidelines* and *System Configuration Framework Reference*).

If you need to add static host entries, you should use Directory Services (described in *Directory Service Framework Reference* , the `dscl` manual page, and "Open Directory and the dscl Tool" (page 72) elsewhere in this document).

Distributing Your Application

Developing an application is only part of the story. You must now get it out there for people to use. Given that this is a UNIX-based operating system, you could just `tar` and `gzip` your file. That's fine if your end users are mostly UNIX users, but this doesn't meet the needs of more general users. To complete the transition, you should package you application like other Mac apps. This chapter walks you through some of those details since they are probably new to you as a UNIX developer.

This chapter is important for all non-command-line developers, whether your application is an end-user commercial suite or an open source tool.

Bundles vs. Installers

Most applications in OS X do not need to use an installer. To make installation and removal easy, OS X provides bundles.

A bundle is basically a directory that contains an application. Unlike normal folders, however, it appears to the user like an ordinary file. The user can double-click a bundle, and the application will launch. Since the bundle is really a directory, it can contain all the support files that are needed for the application.

The reason to use bundles is somewhat obvious if you have installed many applications in OS X: applications in the form of a bundle can be installed simply by dragging the application to the destination folder in the Finder.

There are, however, some situations where bundles are problematic. An application that installs kernel extensions, startup items, system-wide preference panes, or any other system-wide resources cannot be installed by dragging the application, since those resources need to be in a separate place on the drive.

If your application requires installing a startup item, the only practical way to install your application is the use of an installer. OS X makes this easy using PackageMaker. Other commercial solutions are also available from various third parties such as Stuffit InstallerMaker from Aladdin Systems and Installer VISE from MindVision.

In most cases, however, it is preferable to install system-wide components the first time your application is launched. You can do this using Authorization Services, as described in the book *Authorization Services Programming Guide* , available from the Apple Technical Publications website.

For more information about how to create a bundle, see *Bundle Programming Guide* .

Packaging Basics

There are two main applications for compressing your application: Disk Utility (or Disk Copy in older versions of OS X) and PackageMaker. Disk Utility allows you to create compressed disk images (similar to an ISO file, but compressed), whereas PackageMaker creates packages that can be installed using the OS X installer.

The recommended form of application distribution is a compressed disk image. A compressed disk image preserves resource forks that may be present, allows drag-and-drop installation, allows license display, and even allows encryption of data, if required.

If your application is a single application bundle, simply put it and any relevant documentation on a disk image with Disk Utility, then compress it and distribute it.

If you have an application that requires administrator privileges to install into privileged directories or requires more than a simple drag-and-drop installation, use PackageMaker (`/Developer/Applications/PackageMaker`) to build installer packages for Apple's Installer application.

The basics of using Disk Utility to make a disk image are given in the next section. For help using PackageMaker, choose PackageMaker Help from the PackageMaker Help menu.

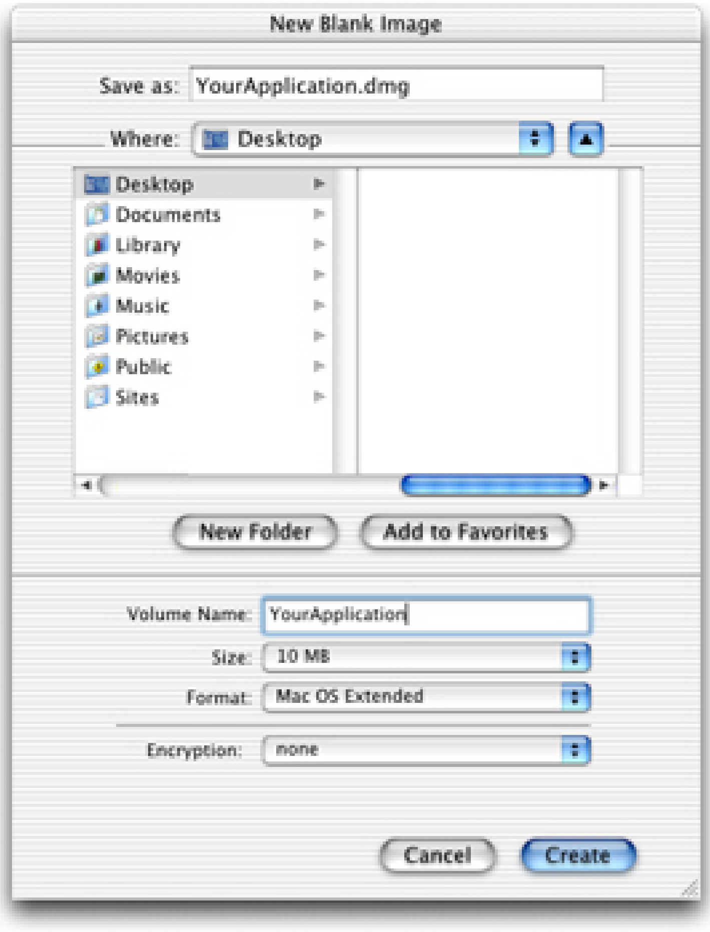
Disk Utility (or Disk Copy)

The following steps help you package your application as a disk image (`.dmg` file) for distribution on OS X.

1. Open `/Applications/Utilities/Disk Utility.app` by double-clicking it.
2. From the Image menu, choose New Blank Image. Disk Utility opens a new window with customization options as in Figure 8-1 (page 66).
3. In the "Save as" text box, enter the name of the compressed file that you will distribute. By default, a `.dmg` suffix is appended to the name you enter. Although it is not required, it is a good idea to retain this suffix for clarity and simplicity.
4. In the Volume Name text field, enter the name of the volume that you want to appear in the Finder of a user's computer. Usually this name is the same as the name of the compressed file without the `.dmg` suffix.
5. In the file browser, set the location to save the file on your computer. This has nothing to do with the installation location on the end user's computer, only where it saves it on your computer.
6. Set the Size pop-up menu to a size that is large enough to hold your application.
7. Leave the Format set to Mac OS Extended (the HFS+ file format).
8. Leave Encryption set to none. If you change it, the end user must enter a password before the image can be mounted, which is not the normal way to distribute an application.

9. Click Create.

Figure 8-1 Disk Utility options



Save as: YourApplication.dmg
Where: Desktop
Volume Name: YourApplication
Size: 10 MB
Format: Mac OS Extended
Encryption: none

Once you have a disk image, mount it by double-clicking it. You can now copy your files to that mounted image. When you have everything on the image that you want, you should make your image read-only. Again from Disk Utility, perform these steps:

1. Unmount the disk image by dragging the volume to the Trash, clicking the eject button next to the device in a Finder window, or selecting the mounted volume and choosing Eject from the Finder's File menu.
2. Choose Convert Image from the Image menu.
3. In the file browser, select the disk image you just modified and click Convert.
4. Choose a location to save the resulting file, change the image format to read-only, and click Convert.

You now have a disk image for your application that is easy to distribute.

Creating Disk Images Programmatically Using hdiutil

If you find yourself regularly creating disk images, you may find it helpful to automate this process. While a complete script is beyond the scope of this document, this section includes a code snippet to get you started.

For more information on hdiutil, see `hdiutil`.

Listing 8-1 Automatic Disk Image Creation using `hdiutil`

```
# Create an initial disk image (32 megs)
hdiutil create -size 32m -fs HFS+ -volname "My Volume" myimg.dmg

# Mount the disk image
hdiutil attach myimg.dmg

# Obtain device information
DEVS=$(hdiutil attach myimg.dmg | cut -f 1)
DEV=$(echo $DEVS | cut -f 1 -d ' ')

# Unmount the disk image
hdiutil detach $DEV

# Convert the disk image to read-only
hdiutil convert myimg.dmg -format UDZO -o myoutputimg.dmg
```

Tell the World About It

Once you have an application, how do you get the word out? First, let Apple know about it. To get your application listed on Apple's main download page for OS X, go to http://www.apple.com/downloads/macosx/submit/ and fill out the appropriate information about your application. You should also go to http://guide.apple.com/ and at the bottom of the page, click Submit a Product to get your application listed in the Apple Guide. You might then also want to send notices to http://www.versiontracker.com/and Macintosh news sites like http://www.macworld.com/ and http://www.macnn.com/.

Additional Features

Although many parts of OS X are the same as other UNIX-based operating systems, OS X also includes many things that set it apart. This chapter lists some of the key details that distinguish OS X from most other UNIX-based operating systems. These may not be important for a basic port of a simple application, but the more robust your application and the more tightly it integrates with the underlying operating system, the more important it is to understand the additional functionality provided by the operating system. Most of the information here is covered only as an overview, with references to more detailed documentation where appropriate.

This chapter is of interest to all varieties of developers, though different developers will have different interests, depending on the nature of your application.

AppleScript and AppleScript Studio

AppleScript is a technology that does for GUI applications what shell scripts do for command-line applications. The scripts control applications using a relatively straightforward messaging protocol (whose commands are application-dependent).

AppleScript Studio is a suite of tools—Xcode, Interface Builder, and the Cocoa application framework—that together allow you to create whole applications written entirely in AppleScript.

For more information on AppleScript and AppleScript Studio, see the AppleScript documentation at http://developer.apple.com/.

Audio Architecture

OS X provides much of the audio functionality normally associated with third-party MIDI and audio toolkits. This gives developers a simple platform for developing dynamic audio applications. For end users, it minimizes the configuration that is normally required to get high-end audio applications to work. As a UNIX developer, it means that if you have been looking for a platform to develop a robust audio application on, you now have one. Among the high-level features of the OS X Core Audio subsystem are these:

- Native multi-channel audio with plug-in support
- Native MIDI

- Audio Hardware Abstraction Layer (HAL)
- A built-in USB class driver compliant with the USB audio specification
- A simplified driver model
- A direct relation with the I/O Kit through the `IOAudioDevice` class that enables rapid device-driver development

If you are developing applications that need access to the audio layer of OS X, you can pursue the extensive resources available at http://developer.apple.com/audio/.

Boot Sequence

The boot sequence in OS X offers a number of advanced features that are not yet common in other UNIX-based and UNIX-like operating systems. These differences are significant enough to warrant their own document: *Daemons and Services Programming Guide* .

In that document, you will learn about legacy startup items (the recommended way to start daemons prior to v10.4), `launchd` (the recommended way in v10.4 and later), and why you should start your daemons with these mechanisms instead of modifying `rc` files.

Configuration Files

Often on a UNIX-based system, you find system configuration files in `/etc`. This is still true in OS X, but configuration information is also found in other places. Networking, printing, and user system configuration details are, by default, regulated by the NetInfo database. Applications usually make use of XML property lists (plists) to hold configuration information. You can view many example property lists by looking in `~/Library/Preferences`.

If changing a configuration file in `/etc` does not have your desired effect, then that information is probably regulated by information in the NetInfo database or by an application's property list.

For example, users and groups are not handled with files in `/etc`. To create a user, you instead should use the Accounts preferences pane or NetInfo Manager. NetInfo Manager can also be used to create groups.

Device Drivers

OS X implements an object-oriented programming model for developing device drivers. This technology is called the I/O Kit. It is a collection of system frameworks, libraries, tools, and other resources. This model is different from the model traditionally found on a BSD system. If your code needs to access any devices other than disks, you use the I/O Kit. I/O Kit programming is done using a restricted subset of C++ that omits features unsuitable for use within a multithreaded kernel. By modeling the hardware connected to an OS X system and abstracting common functionality for devices in particular categories, the I/O Kit streamlines the process of device-driver development. I/O Kit information and documentation is available at http://developer.apple.com/.

The File System

The OS X file system is similar to other UNIX-based operating systems, but there are some significant differences in the file system structure and in case sensitivity. These differences are described in the sections that follow.

File-System Structure and Visibility

The file-system structure of OS X is similar to a BSD-style system. A quick glance at `hier` should comfort you. When in doubt as to where to put things, you can put them where you would in a BSD-style system.There are a few directories that you might not recognize. These differences are described in more detail in "File System Organization" (page 62).

The default behavior of the OS X Finder is to hide the directories that users normally would not be interested in, as well as invisible files, such as those preceded by a dot (`.`). This appearance is maintained by the Finder to promote simplicity in the user interface. As a developer, you might want to see the dot files and your complete directory layout. The `/usr/bin/defaults` tool allows you to override the default behavior of hiding invisible files. To show all of the files that the Finder ordinarily hides, type the following command in the shell:

```
defaults write com.apple.Finder AppleShowAllFiles true
```

Then restart the Finder either by logging out and back in or by choosing Force Quit from the Apple Menu.

There are a couple of other simple ways to view the contents of hidden folders without modifying the default behavior of the Finder itself. You can use the `/usr/bin/open` command or the Finder Go to Folder command. With `open` you can open a directory in the Finder, hidden or not, from the shell. For example, `open /usr/include` opens the hidden folder in a new Finder window. If you are in the Finder and want to see the contents of an invisible folder hierarchy, choose Go to Folder from the Go menu, or just press command-~, and type in the pathname of your desired destination.

For information on how to lay out the directory structure of your completed Mac apps, consult *OS X Human Interface Guidelines* and *Mac Technology Overview*.

Supported File-System Types

OS X supports Mac OS Extended (HFS+), the traditional Macintosh volume format, and the UNIX File System (UFS). HFS+ is recommended and is what most users have their system installed on. Some more server-centric installations have their system installed on UFS. If you develop on UFS, you should thoroughly test your code on HFS+ as well. Although the HFS+ file system preserves case, it is not case sensitive. This means that if you have two files whose names differ only by case, the HFS+ file system regards them as the same file. In designing your application, you should therefore not attempt to put two objects with names that differ only by case in the same directory—for example Makefile and makefile.

However, developing on HFS+ does not necessarily ensure that your application will work on UFS. It is far too easy to write code in which your program opens a file as org.mklinux.formattool.prefs one time and as org.MkLinux.formattool.prefs another time and gets completely different results.

Also, do not assume that a bug is unimportant simply because you have seen it only on UFS. Other file systems have similar properties, including potentially an NFS, SMB, or AFP share, particularly when those shares are being served by something other than a Mac. Thus, a bug that occurs on one file system will likely occur on others.

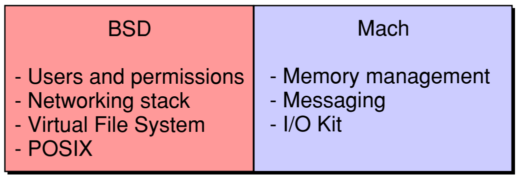
The Kernel

The core of any operating system is its kernel. Though OS X shares much of its underlying architecture with BSD, the OS X kernel, known as XNU, differs significantly. XNU is based on the Mach microkernel design, but it also incorporates BSD features. It is not technically a microkernel implementation, but still has many of the benefits of a microkernel, such as Mach interprocess communication mechanisms and a relatively clean API separation between various parts of the kernel.

Why is it designed like this? With pure Mach, the core parts of the operating system are user space processes. This gives you flexibility, but also slows things down because of message passing performance between Mach and the layers built on top of it. To regain that performance, BSD functionality has been incorporated in the kernel alongside Mach. The result is that the kernel combines the strengths of Mach with the strengths of BSD.

How does this relate to the actual tasks the kernel must accomplish? Figure 9-1 (page 72) illustrates how the kernel's different personalities are manifested.

Figure 9-1 XNU personalities



The Mach aspects of the kernel handle

- Memory management
- Mach messaging and Mach inter process communication (IPC)
- Device drivers

The BSD components

- Manage users and permissions
- Contain the networking stack
- Provide a virtual file system
- Maintain the POSIX compatibility layer

See Inside *Kernel Programming Guide* for more information on why you would (or wouldn't) want to program in the kernel space, including a discussion on the kernel extension (KEXT) mechanism.

Open Directory and the dscl Tool

Open Directory is the built-in OS X directory system that system and application processes can use to store and find administrative information about resources and users. Open Directory includes components such as OpenLDAP and Kerberos for providing local and remote authentication.

By default, each OS X computer runs client and server processes, but the server only serves to the local client. You can also bind client computers to servers other than the local server over a number of protocols including LDAP. Information is then accessed in a hierarchical scheme. In such a scheme, each client computer accesses the union of the information provided first by its local server and then by any higher-level servers it is bound to.

Directory Services is the default way that OS X stores user and some network information. When a user is added, the system automatically adds their information to a local database using Open Directory. Traditional tools such as `adduser` either do not exist or do not work as you might expect. You can add users and groups in several ways:

- Through the Users pane of System Preferences
- Through `/Applications/Utilities/Directory` (for adding groups)
- From the command line (see "Example: Adding a User From the Command Line" (page 73))

You can find more information on NetInfo in the manual pages for `netinfo`, `nidump`, `nicl`, `nifind`, `niload`, `niutil`, and `nireport` on a computer running OS X v10.4 or earlier. NetInfo is no longer supported in OS X v10.5 and later. You should use Directory Service functionality instead.

You can find more information on Directory Service in *Open Directory Programming Guide* and the manual pages `DirectoryService`, `dscl`, `dsconfigldap`, `dsexport`, `dsimport`, and `dsperfmonitor`.

Example: Adding a User From the Command Line

This section shows a simple example of using the Directory Service command-line tool, `dscl`, to add a user to the system. The example specifies some of the properties that you would normally associate with any user.

Note: These commands must be run as the root user. If you are executing them from the command line manually, you should do this with `sudo`. If you are using them in a script, you should use `sudo` when running the script.

1. Create a new entry in the local (`/`) domain under the category `/users`.

   ```
   dscl . -create /Users/portingunix
   ```
2. Create and set the shell property to `bash`.

   ```
   dscl . -create /Users/portingunix UserShell /bin/bash
   ```
3. Create and set the user's full name.

   ```
   dscl . -create /Users/portingunix RealName "Porting Unix Applications To OS X"
   ```
4. Create and set the user's ID.

   ```
   dscl . -create /Users/portingunix UniqueID 503
   ```

   Note: Choosing a user ID is not trivial to do programmatically. For sample code, see the "Starting Points" appendix in *Shell Scripting Primer*.

5. Create and set the user's group ID property.

   ```
   dscl . -create /Users/portingunix PrimaryGroupID 1000
   ```

6. Create and set the user home directory. (Despite the name `NFSHomeDirectory`, this can also be used for a path to a home directory on a local volume.)

   ```
   dscl . -create /Users/portingunix NFSHomeDirectory
   /Network/Servers/techno/Users/portingunix
   ```

   or

   ```
   dscl . -create /Users/portingunix NFSHomeDirectory /Users/portingunix
   ```

7. Set the password.

   ```
   dscl . -passwd /Users/portingunix PASSWORD
   ```

   or

   ```
   passwd portingunix
   ```

8. To make that user useful, you might want to add them to the admin group.

   ```
   dscl . -append /Groups/admin GroupMembership portingunix
   ```

This is essentially what System Preferences does when it makes a new user, but the process is presented here so you can see more clearly what is going on behind the scenes with the database. A look through the hierarchies using `dscl` interactively also helps you understand how the database is organized.

Note: In early versions of OS X, the `nicl` utility served the same purpose. The syntax was similar, with the most notable difference being the use of "/" instead of "." as the first argument.

Printing (CUPS)

OS X, beginning in version 10.2, uses the Common UNIX Printing System (CUPS) as the basis of its printing system. CUPS is a free software (open source) print system that is also popular in Linux and UNIX circles. For more information on CUPS, see the OS X Printing website at http://developer.apple.com/printing.

Bonjour

Bonjour is an implementation of the IETF ZeroConf working draft. Bonjour has three basic parts:

- IP assignment

- Multicast DNS
- Service discovery

You can see the first part, IP assignment, simply by plugging two computers together using a crossover cable and setting them up to use DHCP to obtain an address. Upon failing to obtain an address, the computers choose an IP number randomly from within the range reserved for ZeroConf numbers, then ARP to see if anyone has that address already, repeating this process as needed until a free address is found. This means that a relatively large number of machines can choose their own IP numbers without any administration.

Multicast DNS is a service that allows you to look up the Zeroconf IP number assigned to a particular machine without an established DNS infrastructure. Your computer sends out a request using IP multicast, asking who responds for a particular host name. The host in question then responds. That host could be a computer, a printer, or any other device that might reasonably exist on an IP-based network.

Service discovery is a way to query local machines (using multicast DNS) or a DNS server to see what machines (computers, printers, and so on) support a given service. If you are writing a service, you can register the service using Bonjour. Then anyone else who makes a request sees that your machine supports that service.

The service discovery portion of Bonjour is independent of the IP assignment portion. Thus, Bonjour can also be used for service discovery on a more traditional, configured IP network.

For more information about Bonjour, see *DNS Service Discovery Programming Guide* .

Scripting Languages

With OS X, you can run shell scripts in its native `sh` compatible shell, `bash`, or the included `csh` and `tcsh`. You can also run Ruby, Perl, Python, Tcl, PHP, and other scripts you have developed. In addition, OS X provides an Apple-specific scripting language, AppleScript. Although AppleScript is immensely powerful and can be used to build applications itself, it is designed mainly to communicate with graphical components of the operating system. There are other uses you can find for it, but it is not a replacement for UNIX-style scripting languages. You can use it, though, to put a GUI front end onto your traditional scripts.

AppleScript conforms to the Open Scripting Architecture (OSA) language and can be used from the command line through the `osascript` command. Other languages can be made OSA compliant to enable interaction with the operating system.

Security and Security Services

OS X implements the Common Data Security Architecture (CDSA). If you need to use Authorization Services, Secure Transport, or certificates within the scope of CDSA, online documentation is available at http://developer.apple.com/.

In addition, OS X provides OpenSSL and PAM to ease porting of applications from other UNIX-based operating systems to OS X.

Role-Based Authentication

By default, there is no root user password in OS X. This is a deliberate design decision to maximize security and to simplify the user interface. Your applications should not assume that a user needs superuser access to the system. For power users and developers, `sudo` is provided to run privileged applications in the shell. Privileged applications can also be run by members of the admin group. By default, the admin group is included in the list of "sudoers" (which can be found in the file `/etc/sudoers`). You can assign users to the admin group in System Preferences:

1. Click the Users button.
2. Select a user from the list and click Edit User, or make a new user.
3. Click the Password tab.
4. Check Allow user to administer this computer.
5. That user can now use administrative applications as well as `sudo` in the shell.

Although it is generally considered unsafe practice to log in as the root user, it is mentioned here because the root user is often used to install applications during development. If during development you need to enable the root user for yourself.

In OS X prior to version 10.5:

1. Launch `/Applications/Utilities/NetInfo Manager`.
2. Choose Domain > Security > Authenticate.
3. As prompted, enter your administrator name and password.
4. Now choose Domain > Security > Enable Root User. The first time you do this, you need to select a root password.

In OS X version 10.5 and later:

1. Launch `/Applications/Utilities/Directory Utility`.

2. Click the lock icon to authenticate yourself.
3. Select Edit > Enable Root User. The first time you do this, you need to select a root password.

Alternatively, you can use `sudo passwd root` from the shell and set the appropriate root password.

Important: Do not assume that an end user of your software can enable the root user. This information is provided to help you in development work only; software you distribute should never require the user to enable the root user.

(Re)designing for Portability

When porting applications to OS X, you might consider making architectural changes to make your port (and future ports) more maintainable.

If you are a new developer designing an application that you eventually hope to use on multiple platforms, many of the portability issues described in this chapter apply regardless of the platforms involved.

What is Portability?

There are many different definitions for portability in the context of code design. One definition of portability is limiting yourself to functions specified in a commonly accepted standard such as the Single UNIX Specification (SUS) or the Portable Operating System Interface (POSIX). This is useful for non-GUI software, particularly when moving among UNIX-based and UNIX-like systems, but it does not address graphical user interfaces and does not allow you to take advantage of operating-system–specific capabilities.

The alternative to such limitations is to design your code in a modular fashion so that additional features specific to a given OS can be "plugged into" your application. Whether you do this with a plug-in interface, a class hierarchy, or simple abstraction, this is the most effective way to allow your software to be easily ported to multiple platforms without sacrificing functionality.

There are many different ways to achieve this degree of portability, all of which have valid uses. The goal of this chapter is to describe some of these and to explain when it is appropriate (or inappropriate) to use each one. This should help you avoid some of the common pitfalls when modifying software to support multiple platforms.

Of course, the proper time to make such decisions is before you begin writing the first line of code. Since this is not always possible, this chapter will also describe ways to retrofit these concepts into an existing application in the most maintainable manner possible.

Using Abstraction Layers

Abstraction layers are the easiest, most general-purpose way of making code more portable. The basic concept is well discussed in computer science textbooks. However, most people still find themselves unclear on when it is appropriate to add an additional layer of abstraction.

The most straightforward rule of abstraction is that you should split functionality by use. If a block of code is likely to be reused, even if the code needs to be slightly modified or wrapped in different code, that block should be a separate function. If a block of code is only relevant to the function in which it is enclosed, it probably should not be, unless doing so represents a significant benefit to readability.

You'd be surprised how well this single rule fits most situations. Take file access, for example. Say you want to take advantage of the Carbon file API (to get alias support, for example). The code to open, read, and write a file is used in many places in most applications. Therefore, it should be abstracted into its own function and called from those places.

What you should generally avoid doing, though, is abstracting the open, read, and write calls individually, then using them inside large loops. This leads to needlessly unreadable code with lots of very small abstraction layer functions. You should instead present an interface that makes sense in the context of your application.

For example, if your application performs streaming, you might have the following overall structure:

- A function that opens a file
- A function that reads the next *n* bytes that returns to a buffer containing the data
- A function that rewinds the stream by *n* bytes
- A function that closes the file

Your open function might return an opaque handle that can be passed around within the core of the application but whose structure is unknown outside the file functions.

Similarly, you might have functions that read and write the preferences file using a large data structure, a function to read the header of a file, and so on.

Avoid Conditionalizing Code

Don't fall into the trap of conditionalizing hundreds of bits of code with `#ifdef` directives. (An occasional `#ifdef` is OK.) This quickly leads to unmanageable code, particularly when you support OS X, multiple UNIX-based and UNIX-like systems, Classic Mac OS, and Windows.

If you study the spots in your code that you need to special-case, more often than not, you will find that their issues are closely related. For example, they might all be graphics routines. If so, you might pull all the functions that call graphics routines into their own file and conditionalize the inclusion of the entire file. For example, you might use using one file for X11 routines, one for Win32 routines, and one for Carbon or Cocoa routines. It is easier to conditionalize code in a few core functions than to conditionalize it in a hundred random places in the code. Similarly, it is easier to replace an entire file than to replace bits of a file.

GUI Abstraction Issues

Most *non-GUI* abstraction layers should be as thin as possible, since a thick abstraction layer tends to lead to significant platform divergence. However, with a GUI, platform divergence is desirable and is thus an exception to this rule.

If you want your application to look identical on all platforms, then you should make the abstraction layer thin. However, this tends to result in OS X users throwing your application in a dumpster because it doesn't feel like a Mac application. OS X has common GUI style rules that most X11 applications don't follow.

For example, X11 applications often have per-window menus, while OS X has per-application menus (with the ability to add or remove menus when a different window is in the foreground). X11 applications tend to have very square features and small, space-efficient buttons, while Mac apps tend to have rounded, large, easier-to-read buttons.

Similarly, other operating systems, such as Windows and Classic Mac OS, have different design rules. You must understand the GUI design rules for each computing platform or your application will not be accepted easily on those platforms.

For this reason, it is easiest to split your entire user interface into a separate file or directory for X11, then clone that and rewrite it for Carbon or Cocoa when porting to OS X. That way, you can heavily redesign the OS X interface to match what Mac users expect without annoying your UNIX users.

A good way to implement this is to write your GUI to operate in a separate thread (or multiple threads, if desired). It can then communicate with the core of the application using pipes or other platform-specific solutions.

Another way to implement this is to simply have the GUI call functions in the core of the code. This design is effective when the core of the application is entirely GUI driven, but tends to result in the GUI appearing to wedge if the core of the application can take a long time to complete an operation, and is particularly hard to implement if the core code needs to do something continuously in the background during normal operation. You should consider these issues in your redesign to make your application more palatable to end users.

Other General Rules

Don't fall into the trap of overly abstracting your code. If a block of code will not be used in multiple places, don't abstract it out unless it is platform-specific. Abstracting out code that appears in two or three places is, for readability reasons, usually not worth splitting into a separate function.

Do try to limit your functions to a reasonable length. Abstraction for readability alone is probably not a good idea, as splitting a function unnecessarily can end up making it harder to read. If the purpose of a function doesn't divide in an obvious way into multiple subtasks, it is probably better not to split it. In many cases, though, you may find that an inner loop within that function can be considered a distinct operation unto itself. If so, that loop may be split into a separate function if doing so improves readability.

Don't make abstraction layers more than five or six layers deep, as measured from the top level of a major functional unit. The deeper the nesting, the harder it is to follow the code when debugging.

Don't make functions shorter than about ten lines of code. The readability improvement of such a small change in the outer function is rarely significant. There are, of course, exceptions, such as inline assembly (which should always be abstracted). These exceptions are good candidates for an inline function (or even a macro, if you are so inclined).

Remember that these are guidelines, not rules. You should generally follow your instincts. If you feel like something might be reusable, go ahead and abstract it even if it is only being used in one place. If you feel that splitting an inner loop into its own function is pointless, don't split it. This applies to all guidelines, not just the ones in this chapter.

Using Plug-Ins and Libraries Effectively

Plug-ins can be an effective way to minimize platform-specific code in the core of an application. This section describes places where plug-ins are appropriate. The specifics of writing plug-ins for OS X are described in depth in "Dynamic Libraries and Plug-ins" (page 29).

Effectively managing plug-ins can be tricky when dealing with multiple platforms. You may choose to have a plug-in for each platform, or for each service, or both. The choice of methodology depends largely on the amount of code that you need to put into an external module.

If you have only a few platform-specific bits, it is probably sufficient to have a single per-platform plug-in. A good place for such a design is in an application that needs, for example, to support a single feature in both Mac OS 9 and OS X. By isolating this block of code into an external module, the appropriate piece can be loaded according to the platform in which the application is launched.

Another approach is to have a separate plug-in for each distinct service from the operating system. This is particularly effective if the number of services is significant. Separating services into separate modules makes debugging each service easier from a version control point of view.

Designing a Plug-In Architecture

There are several right ways to design a plug-in architecture and many, many more wrong ways. The most important features of a plug-in architecture are extensibility, simplicity, extensibility, robustness, and extensibility, with emphasis on extensibility.

Ensuring that an architecture is robust is the responsibility of the implementors, and is beyond the scope of any design document. You should consider extensibility and simplicity early and often during the design process.

Simplicity

In general, the simpler the plug-in API, the more likely people will use it rather than grafting hacks into other parts of the code. Of course, this becomes a problem if you fail to expose features that are needed for a given plug-in. To solve this problem, design your API based around message passing concepts rather than function calls. That way, you can easily add additional types of messages to the API without modifying the API itself.

Of course, if your plug-in API only needs to handle one particular type of communication and you are relatively certain that this will not change, a plug-in architecture based on functions is somewhat easier to use. As with all designs, there are tradeoffs.

Extensibility

Extensibility in API design is a tricky issue. When you are creating a plug-in architecture in general, you can rarely envision the sorts of plug-ins that might eventually be useful. Someone might want plug-ins to add additional functionality that no one had even thought of, much less invented, when you created the architecture

You can also use plug-ins to abstract the interface presented by platform-specific services into a more generic form to simplify your application. These designs are more straightforward than architectures intended for adding new functionality; creating plug-in architectures for adding features are beyond the scope of this document.

The first step in designing a plug-in API for service abstraction is to choose the level of abstraction at which the application ends and the plug-in begins. If you choose a level that is too close to the application, the plug-ins will contain redundant code. If you choose a level that is too far removed, you will eventually need to interface your application with an environment that does not meet your initial expectations. As a result, the interface code for that module will become excessively complex. In either case, a rewrite is probably in order, which wastes time and resources that could have been saved had you chosen the right level of abstraction to begin with.

Choosing the level of abstraction for a plug-in interface can range from straightforward to downright impossible, depending on the application. In general, you should choose to split out the smallest amount of code possible such that the following conditions are met:

- No assumptions are made about the data structures of the underlying service.
- Return values are standardized in a way that meets the needs of your application.
- The calling convention can be implemented on any system that provides the support necessary for your application to function—that is, no special features of the underlying service should be implied at the plug-in API level or in your application.
- The data types passed to and from the plug-in API are one of two kinds: either base types in the programming language of choice, or composites of those base types that are structurally relevant to the application as a whole rather than to the underlying service.

Designing a plug-in API in this fashion may, however, result in substantial duplication of code. In such environments, a nested plug-in approach may be more practical.

Nesting Modules for Shared Functionality

The concept of nesting plug-ins is straightforward. You should consider using nested plug-ins when you find numerous opportunities for dividing an application from its plug-ins—that is, when dealing with a general class of underlying services divided into a number of subclasses that contain multiple specific variants with common characteristics.

Nested modules are convenient for:

- Authentication—a generic authentication layer with a generic plug-in for UNIX-based systems and specific plug-ins below to handle platform-specific variations
- Databases—at the top level, you might have ODBC, JDBC, STET, and SQL, with submodules for other, more specific SQL implementations
- Printing—for example, you might have a plug-in that handles PostScript printers with narrower plug-ins that override generic assumptions for a particular printer model

In short, whenever you have a group of underlying services that are substantially similar in behavior, but where you want to be able to also support services with dramatically different behavior, a nested plug-in approach is an effective design (unless the differences in behavior at the lowest level are very minimal).

Example: Database Support

A good example of plug-in API design is database access. As an example, consider a project that uses a database to obtain song playlist information.

Such a program requires various specific pieces of information from the database. For a given song, it might need the title, the artist, the total length, the intro (talk-over) time, the time at which the next song should begin playing (trigger time), and the time at which the song should be faded out, if applicable. In addition, a comments field could be included for composer or other genre-specific information. This is referred to as the internal data level.

To facilitate support for arbitrary databases, you add the first layer of abstraction at the internal data level. The core code calls functions with names like `getsongname` and `getsongtrigger`. Any arbitrary database, regardless of the query language used, can easily return a string or an integer (or at worst, be coerced into doing so). This is considered the minimum level of functionality needed to support this application, and thus forms the first plug-in split.

However, a surprising number of databases use a common syntax, SQL, for making requests. Although the syntax is the same, certain details (data types) are different, and the libraries used for accessing them are also different. For these reasons, supporting multiple SQL servers is a desirable goal, because you can use most common databases as storage.

Because the SQL instructions themselves are so similar between databases, the second split comes somewhat naturally. The code to actually execute a query can be placed in an implementation-specific module, and the code to generate the query—for a song's name, for example—can be placed in a generic SQL language module. The API for the implementation-specific module could, for example, include:

- `getdbc`—Returns a pointer to a database connection object. The SQL core code treats this as an opaque type, but this is necessary to allow many database implementations to be used in a multithreaded environment. This information should not leave the SQL core unless the larger design requires connections to multiple databases simultaneously.
- `opendb`—Opens a database connection.
- `closedb`—Closes a database connection.
- `dbquerystring`—Executes an SQL query and returns the first result as a string.
- `dblistsongs`—Returns an array of song IDs that match an SQL query.

It might strike you as odd that this design doesn't specify a generic SQL query call. This is largely a space–time tradeoff. Implementing a generic query would allow the GUI, for example, to request all of the information about a call in a single request. However, doing so would add a great deal of code for a relatively small time benefit. Since the GUI does not need to refresh frequently, and since the number of queries per second, therefore, tends to be relatively small, it makes sense to design the API to be as simple as possible. If low latency and high bandwidth are required for the specific application, a generic routine is desirable.

If you are familiar with SQL, though, you are probably aware that the similarities among SQL implementations end at inserting new tables into the database. Because this is such a small piece of the overall picture for this type of application (one or two lines of code total), it could easily be "special-cased" in the core code or (preferably) in the generic SQL code. This is one of those cases where you must decide whether the extra 1% of functional abstraction is worth the extra effort involved. In many cases, it is not.

If creating tables is more significant, you could create them in one of two ways: by adding a function in the implementation-specific plug-in, or by adding a function in the generic SQL plug-in, using a table for the data types (which could, if desired, reside in the implementation-specific plug-in).

To demonstrate the effectiveness of this design, MySQL database support was added to an application that used this exact design in about an hour. With the exception of table creation, no modifications to the generic SQL support routines were required.

In summary, proper plug-in design is much like proper abstraction. Code that can be reused should be reused, but the API to the plug-ins should not assume any knowledge of the underlying system.

Command-Line Tool Portability

If your software depends on command-line tools that ship as part of the operating system, you should be aware of differences in command-line flags and behavior. The behavior of some command-line tools varies not only across platforms but also across different versions of OS X.

For an extensive explanation of these differences, see "Designing Scripts for Cross-Platform Deployment".

Architectural Portability

As a good general rule, when writing abstraction layers, plug-in architectures, and so on, architectural portability should be considered. Most existing open source software is already fairly flexible in this regard. However, when dealing with OS-specific code, you may find that consistent support for things like endianness and alignment are not always considered.

When you encounter code specific to OS X with architectural dependencies on a particular processor architecture, there are a number of ways to handle the situation. Here are some tips that should help:

- Altivec or SSE code should be special-cased with equivalent scalar versions that can be compiled in. This will ensure that it is easy for developers familiar with other chip architectures to understand what is happening and write equivalents for other architectures.
- Testing of alignment should generally occur at compile time, not configuration time, to avoid unnecessary problems when cross-compiling.

- Testing of endianness should generally occur at either compile time or execution time, at your option.
- Executing intermediate build products is a bad idea and should be avoided where possible.

For more detailed information, see "Compiling for Multiple CPU Architectures" (page 20) and the document *Universal Binary Programming Guidelines, Second Edition* .

Glossary

ADC See Apple Developer Connection

Apple Developer Connection The primary source for technical and business resources and information for anyone developing for Apple's software and hardware platforms anywhere in the world. It includes programs, products, and services and a website filled with up-to-date technical documentation for existing and emerging Apple technologies. The Apple Developer Connection is at http://www.apple.com/developer/.

Aqua The graphical user interface for OS X.

bom (Bill Of Materials) A file in an installer package used by the Installer to determine which files to install, remove, or upgrade. It contains all the files within a directory, along with information about each file such as the file's permissions, its owner and group, size, its time of last modification, a checksum for each file, and information about hard links.

bundle A directory in the file system that stores executable code and the software resources related to that code. Applications, plug-ins, and frameworks are types of bundles. Except for frameworks, bundles are file packages, presented by the Finder as a single file.

Carbon An application environment for OS X that features a set of programming interfaces derived from earlier versions of the Mac OS. The Carbon API has been modified to work properly with OS X, especially with the foundation of the operating system, the kernel environment. Carbon applications can run in OS X, Mac OS 9, and all versions of Mac OS 8 later than Mac OS 8.1.

Classic An application environment for OS X that lets you run non-Carbon legacy Mac OS software. It supports programs built for both Power PC and 68K chip architectures and is fully integrated with the Finder and the other application environments.

Cocoa An advanced object-oriented development platform for OS X. Cocoa is a set of frameworks with programming interfaces in both Java and Objective-C. It is based on the integration of OPENSTEP, Apple technologies, and Java.

Darwin Another name for the core of the OS X operating system. The Darwin kernel is equivalent to the OS X kernel plus the BSD libraries and commands essential to the BSD command-line environment. Darwin is open source technology.

.dmg file An OS X disk image file.

Finder The system application that acts as the primary user interface for file-system interaction.

HFS (Hierarchical File System) The Mac OS Standard file-system format, used to represent a collection of files as a hierarchy of directories (folders), each of which may contain either files or folders themselves. HFS is a two-fork volume format.

HFS+ The Mac OS Extended file-system format. This file-system format was introduced as part of Mac OS 8.1, adding support for filenames longer than 31 characters, Unicode representation of file and directory names, and efficient operation on very large disks. HFS+ is a multiple-fork volume format.

Mach-O The executable format of Mach object files. This is the default executable format in OS X.

NetInfo The network administrative information database and information retrieval system for OS X. Many OS X services consult the NetInfo database for their configuration information.

nib file An XML archive that describes the user interface of applications built with Interface Builder.

`.pkg` file An OS X Installer file. May be grouped together into a metapackage (`.mpkg`).

plist See property list.

property list A structured, textual representation of data that uses the Extensible Markup Language (XML) as the structuring medium. Elements of a property list represent data of certain types, such as arrays, dictionaries, and strings.

Xcode Apple's graphical integrated development environment. It is available free with the OS X Developer Tools package.

XNU The OS X kernel. The acronym stands for X is Not Unix. XNU combines the functionality of Mach and BSD with the I/O Kit, the driver model for OS X.

Document Revision History

This table describes the changes to *Porting UNIX/Linux Applications to OS X* .

Date	Notes
2012-06-11	Fixed link for Qt.
2012-03-14	Folded in content from Technical Note TN2071.
2010-09-01	Added link to Shell Scripting Primer's adduser and addgroup scripts.
2008-04-08	Fixed minor typographical errors and omissions.
2008-02-08	Corrected instructions for building with an external Makefile target.
2006-11-07	Improved shared library and bundle explanation. Updated example from nicl to dscl.
2006-10-03	Added an explanation of why one should not conditionalize code specifically for OS X.
2006-03-08	Fixed typographical errors.
2006-01-10	Corrected external target build instructions.
2005-10-04	Added hdiutil example and made minor wording improvements.
2005-08-11	Added an explanation of universal binaries. Added a reference to "Dynamic Library Programming Topics."
2005-07-07	Added missing step in packaging instructions. Added link to GNU autoconf. Globally changed "Disk Copy" to "Disk Utility."
2005-04-29	Updated for OS X v10.4.
2004-08-31	Minor wording changes.

Date	Notes
2004-04-22	Minor wording changes.
2003-10-17	Updated for Panther