Technical Note TN2137 Building Universal Binaries from "configure"-based Open Source Projects

This technical note discusses how to build universal binaries out of some "configure"-based Open Source projects.

- Introduction
- Configuring for universal binaries
- Merging multiple builds
- What to Watch Out For
- Document Revision History

Introduction

The Xcode IDE provides a streamlined build system that hides most of the complexity of building a universal binary. Settings, environment variables, and the commands to actually build and link the source file are either implict in pre-set build configurations, or are easily adjustable from the Xcode UI.

Many open source projects use a build-time configuration script to determine the environment in which the program will compile and run, including machine information (such as CPU type, word order and pointer size), and which header files and libraries are available on the system. This script typically constructs one or more Makefiles and header files. The Makefiles will contain the compiler and linker options, and cannot benefit from the automated constructs in the Xcode IDE. The header files generated by this process typically contain constants that are used in feature tests., e.g.

```
#ifdef HAVE_UNISTD_H
# include <unistd.h>
#endif
```

with a `#define HAVE_UNISTD_H` in the generated `config.h` file.

This approach works well in traditional compilation environments, and has made it relatively easy to port many Open Source projects to Mac OS X; the problem arises because the configure environment was not envisioned with a universal binary-like situation in mind.

Note: This technote does not discuss any code changes that might be necessary to migrate an existing project to an Intel-based Macintosh. See the Universal Binary Programming Guide for discussion of which code changes may be required.

There are two main approaches to handling this, and producing a universal binary. We will examine both approaches using the GNU Hello package, a "Hello, World" program designed to show how to use the configure script. The GNU Hello source is available from the Free Software Foundation.

Back to Top

Configuring for universal binaries

The first approach is to simply have `configure` build a universal binary, by passing in the appropriate `CFLAGS` and `LDFLAGS` environment variables. This is done simply by running

```
env CFLAGS="-O -g -isysroot /Developer/SDKs/MacOSX10.4u.sdk -arch i386 -arch ppc" \
  LDFLAGS="-arch i386 -arch ppc"   ./configure --prefix=${HOME}/Hello --disable-dependency-tracking
```

from the shell command prompt. The `--disable-dependency-tracking` option to `configure` causes it to not use gcc's built-in dependency generation code, which does not work with multiple `-arch` targets. To verify that your configure script properly handles the `--disable-dependency-tracking` option, use the help option on configure:

```
./configure --help
```

The help option prints out a message giving usage instructions for configure, as well as arguments it understands how to process.

Note: On an Intel-based Macintosh system the libraries are already universal, and support the Intel and PowerPC architectures, and you may specify only the `-arch i386 -arch ppc` options for `CFLAGS`; on a PowerPC-based Macintosh, you must use the MacOSX10.4u SDK.

After running configure and then building with make, the result is a universal binary in `./src/hello`. We can verify this by querying the file type using the file command:

```
file ./src/hello
```

which would output:

```
src/hello: Mach-O fat file with 2 architectures
src/hello (for architecture i386):      Mach-O executable i386
src/hello (for architecture ppc):       Mach-O executable ppc
```

This executable can be natively run on both PowerPC- and Intel-based Macintoshes.

Back to Top

Merging multiple builds

While the GNU Hello program is one of the most complicated "Hello, World" programs ever written, it is still a relatively simple program: it does not care about byte order, word size, or pointer size; nor does the configuration process generate any executables which themselves produce configuration files based on the machine target. Not all Open Source projects are this simple. For them, there is another approach, which involves using the `lipo` command.

If it had not been possible to configure and build the GNU Hello program using the process above, we could still produce a universal binary by configuring and building the program multiple times (very likely on multiple machines).

On both an Intel- and PowerPC-based Macintosh, configure and build the program as follows:

```
./configure --prefix=${HOME}/Hello
make
```

Copy the resultant `src/hello` programs to a single machine; for example, into `/tmp` on a PowerPC-based Macintosh, with the names `hello-intel` and `hello-ppc`, respectively. Then, use the `lipo` command to combine the two:

```
lipo -create hello-intel hello-ppc -output hello
```

As before, the `file` command can verify the file contents:

```
file hello
```

will report

```
hello: Mach-O fat file with 2 architectures
hello (for architecture i386):  Mach-O executable i386
hello (for architecture ppc):   Mach-O executable ppc
```

For more complicated projects, it may be necessary to install each configuration, and generate a list of Mach-O files (libraries and executables), and run `lipo` on each of them.

Back to Top

What to Watch Out For

Even with this approach, however, there are some things to be aware of:

First, all architectures should be configured the same way. If the configuration for one architecture expects run-time files to be in `/usr/local/etc`, but the configuration for another architecture expects them to be in `/etc`, the run-time behaviors will be different.

Second, sometimes it is not possible to avoid that situation, e.g., if the configuration process uses the processor type (e.g., "ppc" or "i386" or "i686") in the run-time directories. When building such a project for binary release, it may be necessary to manually examine the trees created by the install process, and ensure that either all variants are in the final release, or that symbolic links are used to emulate this. (E.g., `/usr/local/myproj/etc/i386/input.conf` and `/usr/local/myproj/etc/ppc/input.conf` should either both be present, or a symbolic link may be used to point `/usr/local/myproj/etc/ppc` at `/usr/local/myproj/etc/i386`.)

Back to Top

Document Revision History

Date	Notes
2006-10-05	Corrected section on LDFLAGS. Added information on handling configure scripts that don't respond to --disable-dependency-tracking
2005-08-25	Describes some methods for building some existing "configure"-based Open Source packages as universal binaries.

Posted: 2006-10-05
