Converting Static Libraries to Dynamic Libraries for iOS

This is a tutorial that will demonstrate two methods for converting a static library into a dynamic library for use on iOS and OS X. If you are attempting to do this with a library that is available via CocoaPods, then I recommend that you use CocoaPods to do this work for you. The methods I am going to describe should only be used if there are no other options.

##What are static and dynamic libraries? First thing to understand is the differences between using static and dynamic libraries. I have a previous blog post that goes into detail about this. You should be aware that there have been some performance impacts with loading many dynamic libraries when an app gets launched. This is due to the need for the `amfi` daemon to read and validate the contents of the library files to ensure their code signatures match the signing identify of the app that is asking for them to be loaded. Once that is done the library can be safely opened by the dynamic linker and loaded into memory for the app.

For the methods, we are going to use a sample library that was used in the linked post above:

foo.h

```
#ifndef __foo__bar__
#define __foo__bar__

#include <stdio.h>

int fizz();

#endif /* defined(__foo__bar__) */
```

foo.c

```
#include "foo.h"
#include <CoreFoundation/CoreFoundation.h>

int fizz() {
    CFShow(CFSTR("buzz"));

    return 0;
}
```

To follow along with the example, I am going to generate a static library from the library code by calling:

```
$ clang -c foo.c -o foo.o
$ libtool -static foo.o -o libfoo.a -framework CoreFoundation
```

##Method 1: Manual Conversion The scenario is that you get a static library as part of a third-party SDK you have to integrate and you are currently building your app for iOS 8 and above. You would prefer to dynamically link this library to your app, as that is how you are building all of your dependency code now.

First step is to identify the type of library that you are working with. Most static libraries that are distributed are going to be “fat”. This means that they contain code for multiple architectures in them. You can identify these library files by using the `lipo` command.

```
$ lipo -info libfat.a
Architectures in the fat file: libfat.a are: x86_64 arm64
```

You will have to extract each architecture slice from the “fat” library file. I recommend doing this to a new subdirectory along-side the “fat” library you are working with, this is to make later steps easier. To extract a particular architecture slice, we are going to use the `lipo` tool again:

```
$ mkdir -p x86_64
$ lipo libfat.a -extract x86_64 -output ./x86_64/libfoo.a
```

Once you have performed that command, you can verify the extracted file is the correct architecture slice:

```
$ lipo -info ./x86_64/libfoo.a
input file libfoo.a is not a fat file
Non-fat file: libfoo.a is architecture: x86_64
```

Now you should be working with an archive file, this can be verified through the `file` command:

```
$ file ./x86_64/libfoo.a
libfoo.a: current ar archive random library
```

For the next step we are going to enter the same directory as the architecture slice we extracted. Then we are going to use the `ar` utility to extract all of the object files in the library archive. When using the `-x` flag for extraction, the utility will write all of the object files in the library archive to the **current** directory. This is why I recommend creating separate directories for each architecture slice you plan on converting.

```
$ cd x86_64
$ ar -x libfoo.a
```

This will extract all the compiled executable code out of the archiv into the individual object files. With these object files, you can use the linker to build a new dynamic library.

```
$ libtool -dynamic *.o -o libfoo_dynamic-x86_64.dylib -framework CoreFo
```

The command above uses `libtool` to build a dynamic library from all of the to build a dynamic library from all of the `.o` object files and to give the created library `libfoo_dynamic-x86_64.dylib`. You will need to supply any additional `-framework` or `-l` flags here that the library normally relied on. This is because these flags will no longer be used to link against the app, they will now link against the new dynamic library you have created.

You have now converted a static library to a dynamic library. You should repeat this process for each of the architecture slices in the original static library file. Once you have completed that, you can use the following `lipo` command to create a new “fat” dynamic library:

```
$ lipo -create ./x86_64/libfoo_dynamic-x86_64.dylib ./arm64/libfoo_dyna
```

This new “fat” dynamic library can now be used as to be linked against as part of your iOS or OS X application. If you encounter any linker errors where the linker cannot resolve symbols with the newly created dynamic library, then you will have to use the second method to convert the static library to a dynamic library.

##Method 2: Wrapping The method of wrapping a static library to use it as a dynamic library is by far the most reliable and easy method to do.

1. Create a new project file in Xcode. This is going to contain the dynamic framework that will “wrap” the static library.

2. Choose a “Cocoa Touch Framework” as the target type and complete the new project and target creation process.
3. Add the static library that you want to convert to the project. You should also add any headers and resources that come with the static library/ framework.
4. Under the “Build Settings” of the dynamic library target:
    - Add the `-all_load` flag to “`OTHER_LDFLAGS`”.
    - Add the path to the static library to the `FRAMEWORK_SEARCH_PATH` and `LIBRARY_SEARCH_PATH` fields.
    - If the static library doesn’t contain slices for specific architectures, the linker will raise a warning about it. You can silence this type of warning by using the `-no_arch_warnings` to “`OTHER_LDFLAGS`”.
5. Under “Build Phases” of the dynamic library target:
    - Add the static library to the “Link Binary With Libraries” phase.
    - Add the headers you want to have exported as public to the “Headers” phase.
    - Add the resources you want to be included with the framework to the “Copy Bundle Resources” phase.
6. Build the dynamic framework target
7. There may be a bunch of build errors that occur. Many of these will be missing symbol errors coming from the linker. To correctly resolve this you will need to go back to “Build Phases” and expand the “Link Binary With Libraries” phase again. Here you will have to add all the system libraries that the static library expects to be linked with. The third-party library should give you a list of what libraries will be needed.
8. Build the dynamic framework again, this time it should be able to succeed.

This should produce a dynamic framework that includes all the necessary code of the static library. This method is going to be necessary for some libraries that don’t export some of the symbols that they use and thus make the first method not viable. The second method is also a nicer way to abstract the libraries that you use.
