

Using CGO with Pkg-Config And Custom Dynamic Library Locations

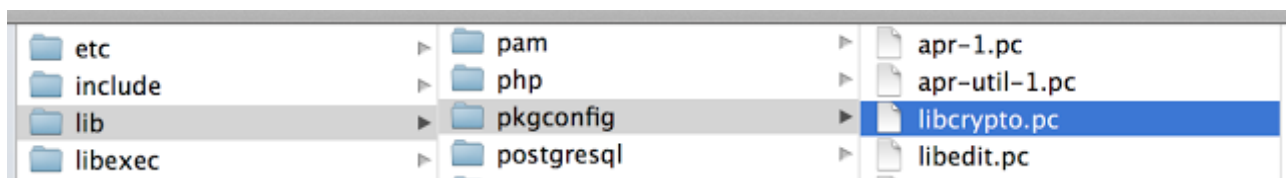
Aug 31, 2013

Earlier in the month I wrote a post about using [C Dynamic Libraries in Go Programs](#). The article built a dynamic library in C and created a Go program that used it. The program worked but only if the dynamic library was in the same folder as the program.

This constraint does not allow for the use of the **go get** command to download, build and install a working version of the program. I did not want to have any requirements to pre-install dependencies or run extra scripts or commands after the call to **go get**. The Go tool was not going to copy the dynamic library into the bin folder and therefore I would not be able to run the program once the **go get** command was complete. This was simply unacceptable and there had to be a way to make this work.

The solution to this problem was twofold. First, I needed to use a package configuration file to specify the compiler and linker options to CGO. Second, I needed to set an environment variable so the operating system could find the dynamic library without needing to copy it to the bin folder.

If you look, you will see that some of the standard libraries provide a package configuration (.pc) file. A special program called **pkg-config** is used by the build tools, such as gcc, to retrieve information from these files.



If we look in the standard locations for header files, **/usr/lib** or **/usr/local/lib**, you will find a folder called **pkgconfig**. Package configuration files that exist in these locations can be found by the **pkg-config** program by default.

```

prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include

Name: OpenSSL-libcrypto
Description: OpenSSL cryptography library
Version: 0.9.8j
Requires:
Libs: -L${libdir} -lcrypto -lz
Cflags: -I${includedir}

```

Look at the **libcrypto.pc** file and you can see the format and how it provides compiler and linker information.

This particular file is nice to look at because it includes the bare minimum format and parameters that are required.

To learn more about these files read this web page: www.freedesktop.org/wiki/Software/pkg-config

The prefix variable at the top of the file is very important. This variable specifies the base folder location where the library and include files are installed.

Something very important to note is that you **can't use an environment variable to help specify a path location**. If you do, you will have problems with the build tools locating any of the files it needs. The environment variables end up being provided to the build tools as a literal string. Remember this for later because it is important.

Run the **pkg-config** program from a Terminal session using these parameters:

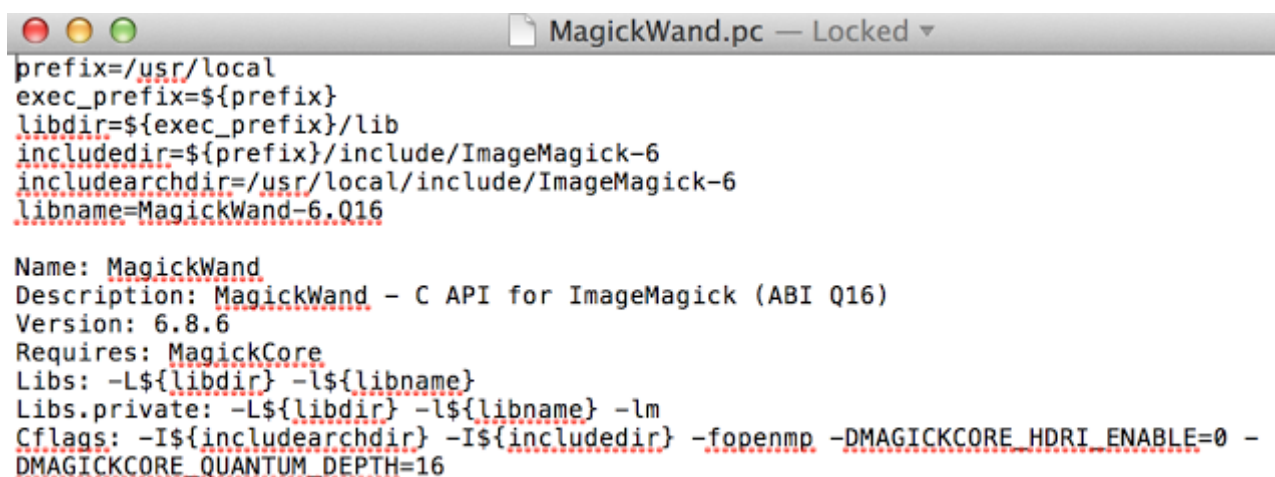
```
pkg-config --cflags --libs libcrypto
```

These parameters ask the **pkg-config** program to show the compiler and linker settings specified in the .pc file called libcrypto.

This is what should be returned:

```
-lcrypto -lz
```

Let's look at one of the package configuration files from ImageMagick that I downloaded and installed under **/usr/local** for a project I am working on:



```

prefix=/usr/local
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include/ImageMagick-6
includearchdir=/usr/local/include/ImageMagick-6
libname=MagickWand-6.Q16

Name: MagickWand
Description: MagickWand - C API for ImageMagick (ABI Q16)
Version: 6.8.6
Requires: MagickCore
Libs: -L${libdir} -l${libname}
Libs.private: -L${libdir} -l${libname} -lm
Cflags: -I${includearchdir} -I${includedir} -fopenmp -DMAGICKCORE_HDRI_ENABLE=0 -
DMAGICKCORE_QUANTUM_DEPTH=16
  
```

This file is a bit more complex. You will notice it specifies that the MagickCode library is also required and specifies more flags such as environmental variables.

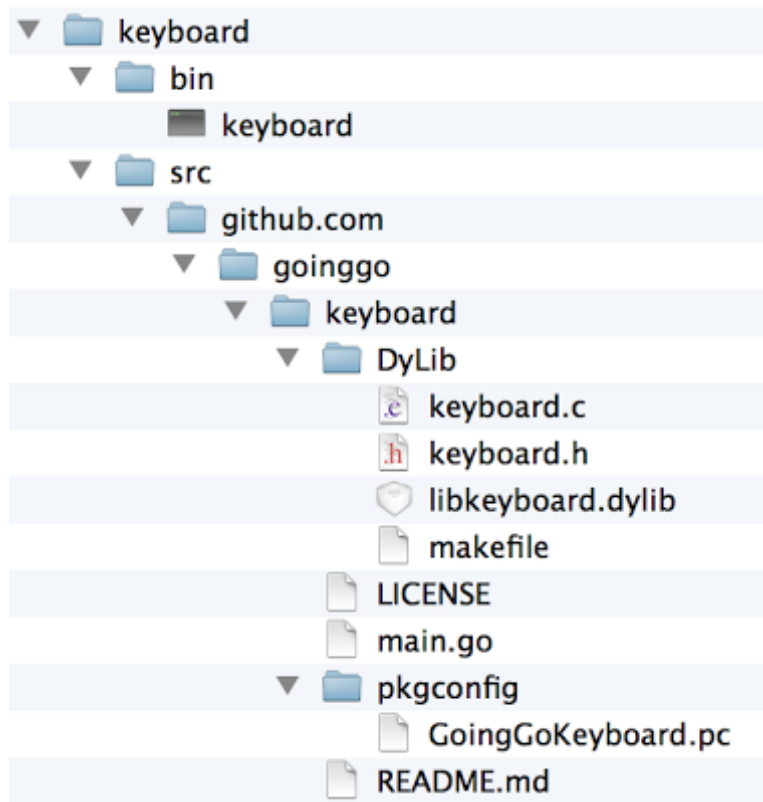
pkg-config --cflags --libs MagickWand

You can see that the path locations for the header and library files are fully qualified paths. All of the other flags defined in the package configuration file are also provided.

Before we begin I must apologize. The dynamic library that I built for this project will only build on the Mac. Read the post I just mentioned to understand why. A pre-built version of the dynamic library already exists in version control. If you are not working on a Mac, the project will not build properly, however all the ideas, settings and constructs still apply.

```
cd $HOME
export GOPATH=$HOME/keyboard
export PKG_CONFIG_PATH=$GOPATH/src/github.com/goinggo/keyboard/pkgconfig
export DYLD_LIBRARY_PATH=$GOPATH/src/github.com/goinggo/keyboard/DyLib
go get github.com/goinggo/keyboard
```

After you run these commands, you will have all the code from the GoingGo keyboard repository downloaded under a subfolder called keyboard inside your home directory.



You will notice the Go tool was able to download, build and install the keyboard program. Even though the header file and dynamic library was not located in the default **/usr** or **/usr/local** folders.

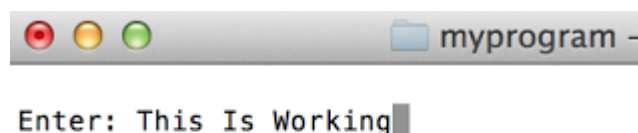
In the bin folder we have the single executable program without the dynamic library. The dynamic library is only located in the DyLib folder.

There is a new folder in the project now called pkgconfig. This folder contains the package configuration file that makes this all possible.

The main.go source code file has been changed to take advantage of the new package configuration file.

If we immediately switch to the bin folder and run the program, we will see that it works.

```
cd $GOPATH/bin
./keyboard
```



When you start the program, it immediately asks you to enter some keys. Type a few letters and then hit the letter q to quit the program.

This is only possible if the OS can find all the dynamic libraries this program is dependent on.

Let's take a look at the code changes that make this possible. Look at the main.go source code file to see how we reference the new package configuration file.

This is the original code from the first post. In this version I specified the compiler and linker flags directly. The location of the header and dynamic library are referenced with a relative path.

```
package main

/*
#cgo CFLAGS: -I../DyLib
#cgo LDFLAGS: -L. -lkeyboard
#include <keyboard.h>
*/
import "C"
```

This is the new code. Here I tell CGO to use the **pkg-config** program to find the compiler and linker flags. The name of the package configuration file is specified at the end.

```
package main

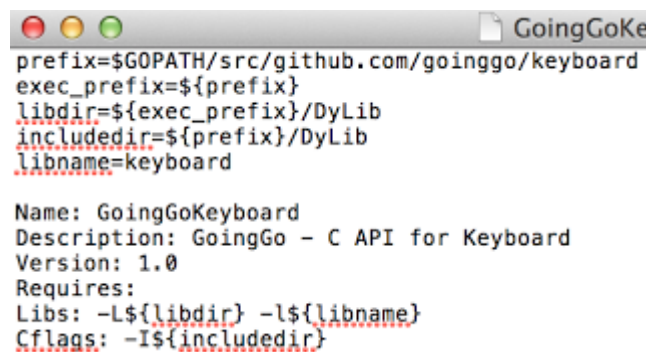
/*
#cgo pkg-config: --define-variable=prefix=. GoingGoKeyboard
#include <keyboard.h>
*/
import "C"
```

Notice the use of the **pkg-config** program option **--define-variable**. This option is the trick to making everything work. Let's get back to that in a moment.

Run the **pkg-config** program against our new package configuration file:

```
pkg-config --cflags --libs GoingGoKeyboard

-I$GOPATH/src/github.com/goinggo/keyboard/DyLib
-L$GOPATH/src/github.com/goinggo/keyboard/DyLib -lkeyboard
```



```

prefix=$GOPATH/src/github.com/goinggo/keyboard
exec_prefix=${prefix}
libdir=${exec_prefix}/DyLib
includedir=${prefix}/DyLib
libname=keyboard

Name: GoingGoKeyboard
Description: GoingGo - C API for Keyboard
Version: 1.0
Requires:
Libs: -L${libdir} -l${libname}
Cflags: -I${includedir}

```

If you look closely at the output from the call, you will see something that I told you was wrong. The `$GOPATH` environment variable is being provided.

Open the package config file which is located in the `pkgconfig` folder and you will see the **pkg-config** program doesn't lie. Right there at the top I am setting the prefix variable to a path using `$GOPATH`. So why is everything working?

Now run the command again using the same **—define-variable** option we are using in `main.go`:

```
pkg-config --cflags --libs GoingGoKeyboard --define-variable=prefix=.
```

```

-I./DyLib
-L./DyLib -lkeyboard

```

Do you see the difference? In the first call to the **pkg-config** program we get back paths that have the literal `$GOPATH` string because that is how the prefix variable is set. In the second call we override the value of the prefix variable to the current directory. What we get back is exactly what we need.

Remember this environment variable that we set prior to using the Go tool?

```
PKG_CONFIG_PATH=$GOPATH/src/github.com/goinggo/keyboard/pkgconfig
```

The `PKG_CONFIG_PATH` environment variable tells the **pkg-config** program where it can find package configuration files that are not located in any of the default locations. This is how the **pkg-config** program is able to find our `GoingGoKeyboard.pc` file.

The last mystery to explain is how the OS can find the dynamic library when we run the program. Remember this environment variable that we set prior to using the Go tool?

```
export DYLD_LIBRARY_PATH=$GOPATH/src/github.com/goinggo/keyboard/DyLib
```

The `DYLD_LIBRARY_PATH` environment variable tells the OS where else it can look for dynamic libraries.

Installing your dynamic libraries in the **/usr/local** folder keeps things simple. All of the build tools are configured to look in this folder by default. However, using the default locations for your custom or third party libraries require extra steps of installation prior to running the Go tools. By using a package configuration file

and passing the **pkg-config** program the options it needs, Go with CGO can deploy builds that will install and be ready to run instantly.

Something else I didn't mention is that you can use this technique to install 3rd party libraries that you may be trying out in a temp location. This makes it real easy to remove the library if you decide you don't want to use it.

If you want to play with the code or concepts on a Windows or Ubuntu machine, read [C Dynamic Libraries in Go Programs](#) to learn how to build your own dynamic libraries that you can experiment with.