

# Using C Dynamic Libraries In Go Programs

Aug 20, 2013

My son and I were having fun last weekend building a console based game in Go. I was recreating a game from my youth, back when I was programming on a Kaypro II.



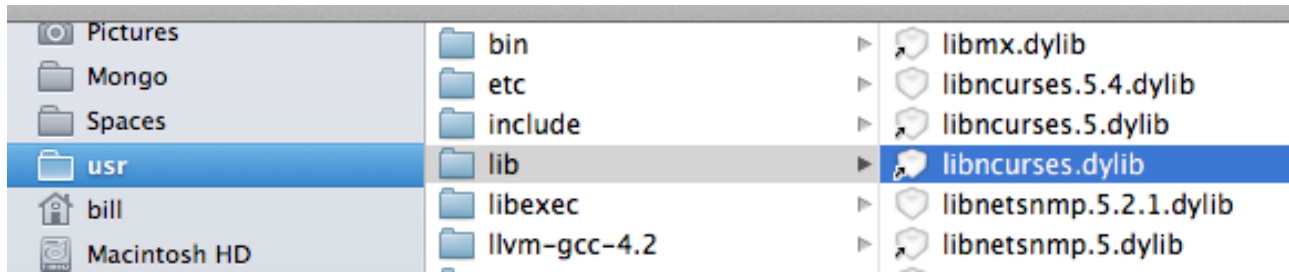
I loved this computer. I would write games in BASIC on it all day and night. Did I mention it was portable. The keyboard would strap in and you could carry it around. LOL.

But I digress, back to my Go program. I figured out a way to use the VT100 escape character codes to draw out a simple screen and started programming some of the logic.

Then something horrible happened and I had a major flashback. I could not get input from stdin without hitting the enter key. Ahhhhh I spent all weekend reading up on how to make this happen. I even found two Go libraries that had support for this but they didn't work. I realized that if I was going to make this happen I needed to build the functionality in C and link that to my Go program.

After a 4 hour coding session at the local Irish pub, I figured it out. I would like to thank Guinness for the inspiration and encouragement I needed. Understand that for the past 10 years I have been writing windows services in C#. For 10 years before that I was writing C/C++ but on the Microsoft stack. Everything I was reading: gcc, gco, static and shared libraries on the Mac and Linux, etc, was foreign to me. I had a lot to learn and still do.

After all my research it became clear I needed to use the ncurses dynamic library. I decided to write a simple program in C using the library. If I could make it work in a compiled C program, I was sure I could get it to work in Go.



The ncurses library on the Mac is located in /usr/lib. Here is a link to the documentation:

<https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man3/ncurses.3x.html>

Here is the C header file code for the test program:

#### **test.h**

```
int GetCharacter();
void InitKeyboard();
void CloseKeyboard();
```

And now the code for the C source file:

#### **test.c**

```
#include <curses.h>
#include <stdio.h>
#include "test.h"

int main() {
    InitKeyboard();

    printf("\nEnter: ");
    refresh();

    for (;;) {
        int r = GetCharacter();
        printf("%c", r);
        refresh();

        if (r == 'q') {
            break;
        }
    }

    CloseKeyboard();

    return 0;
}
```

```
void InitKeyboard() {
    initscr();
    noecho();
    cbreak();
    keypad(stdscr, TRUE);
    refresh();
}

int GetCharacter() {
    return getch();
}

void CloseKeyboard() {
    endwin();
}
```

Now the hard part. How do I build this program using the gcc compiler? I want to make sure I am using the same compiler that Go is using. I also want to make sure I am using the bare minimum parameters and flags.

After about an hour of researching, I came up with this makefile. I told you, I have never done this before.

### makefile

build:

```
rm -f test
gcc -c test.c
gcc -lncurses -r/usr/lib -o test test.o
rm -f *.o
```

When you run the **make** command it will look for a file called makefile in the local directory and execute it. Be aware that each command is one (1) TAB to the right. If you use spaces you will have problems. Obviously you can run these command manually as well. I built this makefile for convenience.

Let's break down the gcc compiler calls from the makefile:

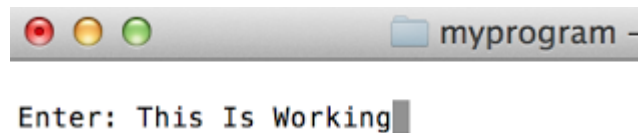
This call to gcc is creating an object file called test.o from the source file test.c. The **-c** parameter tells gcc to just compile the source file and create an object file called test.o

```
gcc -c test.c
```

This second call to gcc links the **test.o** object file together with the shared dynamic library **libncurses.dylib** to create the test executable file. The **-l** (minus lowercase L) parameter is telling gcc to link the libncurses.dylib file and the **-r** (minus lowercase R) parameter is telling gcc where it can find the library. The **-o** (minus lowercase O) parameter tells gcc to create an executable output file called test and finally we tell gcc to include test.o in the link operation.

```
gcc -lnurses -r/usr/lib -o test test.o
```

Once these two gcc commands run we have a working version of the program called test. You must run this program from a terminal window for it to work. To execute the program type **./test** from the terminal window:



Here I started typing letters which are being displayed by the call to printf inside the for loop.

As soon as I hit the letter 'q', the program terminates.

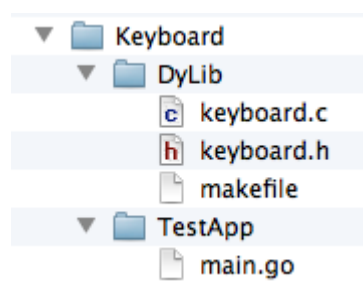
I now have a working version of a program that uses the ncurses dynamic library I want to use in my Go program. Now I need to find a way to wrap these calls into a dynamic library that I can link to from Go.

I was very fortunate to find these web pages which brilliantly show everything we need to know to create shared and dynamic libraries:

[http://www.adp-gmbh.ch/cpp/gcc/create\\_lib.html](http://www.adp-gmbh.ch/cpp/gcc/create_lib.html)

<http://stackoverflow.com/questions/3532589/how-to-build-a-dylib-from-several-o-in-mac-os-x-using-gcc>

Let's work together on making all this work in Go. Start with setting up a Workspace for our new project:



I have created a folder called Keyboard and two sub-folders called DyLib and TestApp.

Inside the DyLib folder we have our C based dynamic library code with a makefile. Inside the TestApp folder we have a single go source code file to test our Go integration with the new dynamic library.

Here is the C header file for the dynamic library. It is identical to the C header file I used in the test application.

**keyboard.h**

```
int GetCharacter();  
void InitKeyboard();  
void CloseKeyboard();
```

Here is the C source file that implements those functions. Again it is identical to the C source file from the test application without the main function. We are building a library so we don't want main.

### **keyboard.c**

```
#include <curses.h>  
#include "keyboard.h"  
  
void InitKeyboard() {  
    initscr();  
    noecho();  
    cbreak();  
    keypad(stdscr, TRUE);  
    refresh();  
}  
  
int GetCharacter() {  
    return getch();  
}  
  
void CloseKeyboard() {  
    endwin();  
}
```

Here is the makefile for creating the dynamic library:

### **makefile**

dynamic:

```
rm -f libkeyboard.dylib  
rm -f ../TestApp/libkeyboard.dylib  
gcc -c -fPIC keyboard.c  
gcc -dynamiclib -Icurses -r/usr/lib -o libkeyboard.dylib keyboard.o  
rm -f keyboard.o  
cp libkeyboard.dylib ../TestApp/libkeyboard.dylib
```

shared:

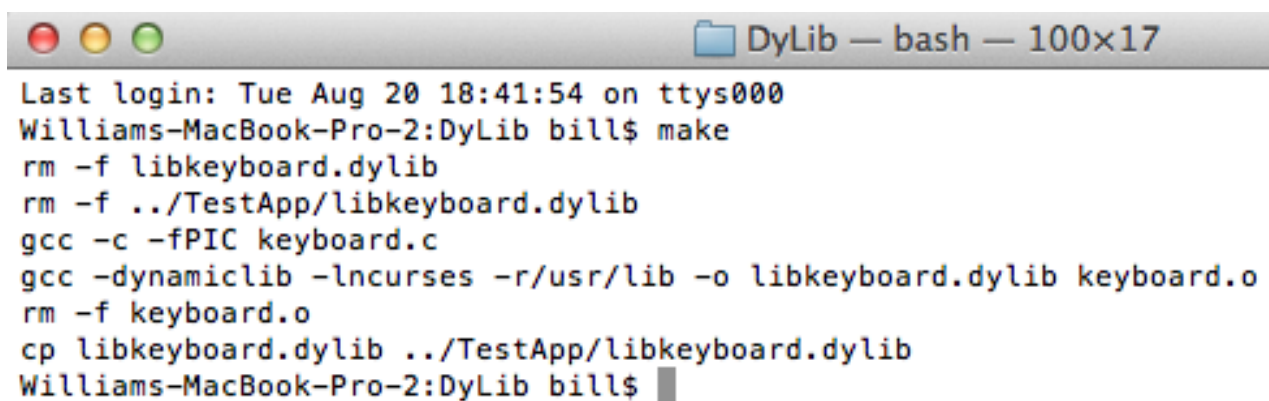
```
rm -f libkeyboard.so  
rm -f ../TestApp/libkeyboard.so  
gcc -c -fPIC keyboard.c  
gcc -shared -W1 -Icurses -r/usr/lib -soname,libkeyboard.so -o libkeyboard.so keyboard.o  
rm -f keyboard.o  
cp libkeyboard.so ../TestApp/libkeyboard.so
```

With this make file you can build either a dynamic library or shared library. If you just run the make command without any parameters, it will execute the dynamic set of commands. To create the shared library run make passing 'shared' (without quotes) as a parameter.

The important flag to notice is **-fPIC**. This flag tells gcc to create position independent code which is necessary for shared libraries. We did not include this flag when we built the executable program.

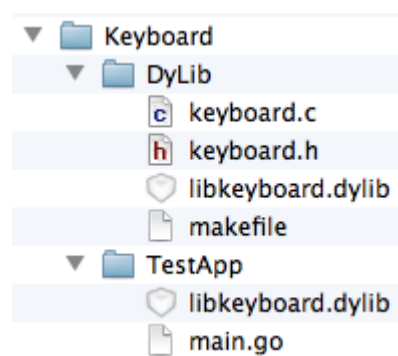
We are going to use the dynamic library moving forward. Mainly because on the Mac this is the most common format. Also, if we clean our Go project later in LiteIDE, it won't remove the file along with the binary. LiteIDE will remove shared libraries on the call to clean.

Let's create the dynamic library by running the make file:



```
DyLib — bash — 100x17
Last login: Tue Aug 20 18:41:54 on ttys000
Williams-MacBook-Pro-2:DyLib bill$ make
rm -f libkeyboard.dylib
rm -f ../TestApp/libkeyboard.dylib
gcc -c -fPIC keyboard.c
gcc -dynamiclib -lcurses -r/usr/lib -o libkeyboard.dylib keyboard.o
rm -f keyboard.o
cp libkeyboard.dylib ../TestApp/libkeyboard.dylib
Williams-MacBook-Pro-2:DyLib bill$
```

We call the make command and it runs the dynamic section of the make file successfully. Once this is done we now have our new dynamic library.



Now we have a new file in both the DyLib and TestApp folders called libkeyboard.dylib.

One thing I forgot to mention is that our dynamic and shared libraries must start with the letters lib. This is mandatory for things to work correctly later. Also the library will need to be in the working folder for the program to load it when we run the program.

Let's look at the Go source code file for our test application:

```
package main

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

func main() {
    C.InitKeyboard()

    fmt.Printf("\nEnter: ")

    for {
        r := C.GetCharacter()

        fmt.Printf("%c", r)

        if r == 'q' {
            break
        }
    }

    C.CloseKeyboard()
}
```

The Go team has put together these two documents that explain how Go can incorporate C code directly or use libraries like we are doing. It is really important to read these documents to better understand this Go code:

<http://golang.org/cmd/cgo/>

[http://golang.org/doc/articles/c\\_go\\_cgo.html](http://golang.org/doc/articles/c_go_cgo.html)

If you are interested binding into C++ libraries, then SWIG (Simplified Wrapper and Interface Generator) is something you need to look at:

<http://www.swig.org/>

<http://www.swig.org/Doc2.0/Go.html>

We will leave SWIG for another day. For now let's break down the Go source code.

```
package main
```

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

In order to provide the compiler and linker the parameters it needs, we use these special cgo commands. They are always provided inside a set of comments and must be on top of the import "C" statement. If there is a gap between the closing comment and the import command you will get compiler errors.

Here we are providing the Go build process flags for the compiling and linking of our program. CFLAGS provides parameters to the compiler. We are telling the compiler it can find our header files in the SharedLib folder. LDFLAGS provide parameters to the linker. We are providing the linker two parameters, -L (minus capital L) which tells the linker where it can find our dynamic library and -l (minus lowercase L) which tells the linker the name of our library.

Notice when we specify the name of our library it does not include the lib prefix or the extension. It is expected that the library name starts with lib and ends in either .dylib or the .so extensions.

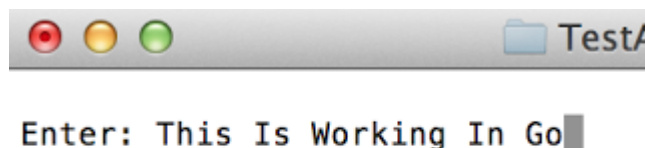
Last we tell Go to import the pseudo-package "C". This pseudo-package provides all the Go level support we need to access our library. None of this is possible without it.

Look at how we call into the each of our functions from the library:

```
C.InitKeyboard()  
r := C.GetCharacter()  
C.CloseKeyboard()
```

Thanks to the pseudo-package "C" we have function wrappers for each function from the header file. These wrappers handle the marshaling of data in and out of our functions. Notice how we can use a native Go type and syntax to get the character that is entered into the keyboard.

Now we can build the test application and run it from a terminal session:





Awesome. Working like a champ.

Now my son and I can continue building our game and get the keyboard action we need to make the game really fun.

It has taken me quite a few number of hours to get a handle on all of this. There is still a lot to learn and support for this will only get better. At some point I will look at SWIG to incorporate C++ object oriented libraries. For now, being able to bring in and leverage C libraries is awesome.

If you want to see and access the code, I have put it up in the GoingGo github repository under Keyboard. Have Fun !!

Read Part II: [Using CGO with Pkg-Config And Custom Dynamic Library Locations](#)