I thought it was worth putting this together as several people have asked me now. If you’re interested in hacking on Docker on Windows, this describes how I set-up my development environment and perform many development workflows.

While some colleagues tend to make their environment look as much like Linux as they can, I tend to keep where possible to using a native Windows tools whenever possible.

I want to emphasis this is not a definitive guide. There isn’t a “correct” development environment. That’s very particular to an individual. However, hopefully this guide will get you up and running if you’re scratching your head on where to start.

For this guide, I’m going to setup a clean Windows 10 64-bit machine as if it were my actual development box. I’m using a fully up to date build of Windows with all Windows updates applied. I could equally as well have used Windows Server 2016.

There will be a couple of other machines required as I’m setting up a client machine. Particularly a Windows Server 2016 machine to support “Windows Server Containers” (client only supports “Hyper-V Containers”). And a Linux machine configured to run Docker.

The Windows Server 2016 and Linux machines will be used as test machines.

# Configure the dev machine to run containers

A good place to start here is <https://msdn.microsoft.com/en-us/virtualization/windowscontainers/quick_start/quick_start_windows_10>. This just sets a baseline. I won’t repeat the steps here, but you’re essentially enabling containers and Hyper-V , installing docker, and configuring docker to run as a Windows service.

Note if your dev machine itself is a Hyper-V VM, you will need to enable nested virtualization for that VM. When the VM is turned off, run the following on the host system:

Set-VMProcessor <vmname> -ExposeVirtualizationExtensions $true

Once you’ve got docker up and running, you’ll actually be disabling it, as the most likely thing you’d want to run in a development scenario is the version of docker you are hacking on. So if the service is running, simply run

* stop-service docker
* dockerd --unregister-service
* delete the docker.exe and dockerd.exe binaries you downloaded
* Remove the path they were in from the path.

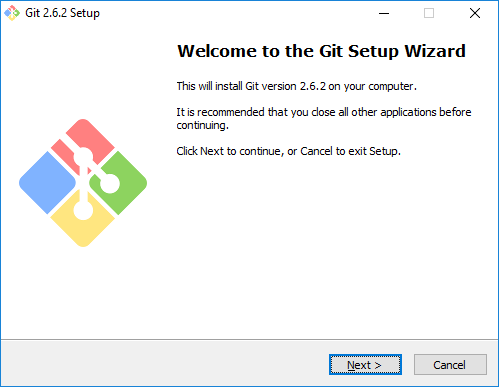
# Configure the Windows test machine to run containers

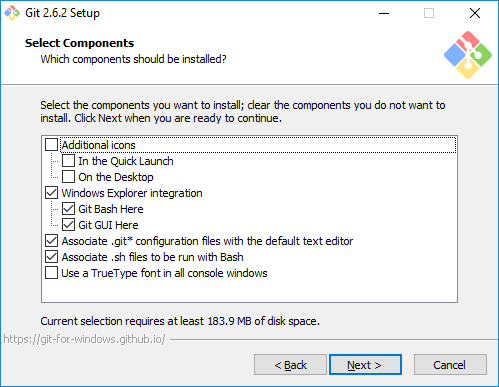
Again, follow the online documentation. If

# Things to install

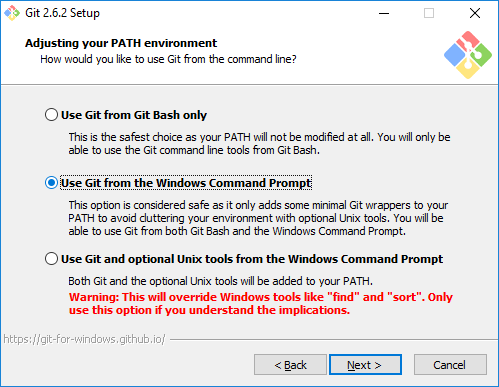
## Git for Windows.

Use the 64-bit installer from <https://git-scm.com/download/win>

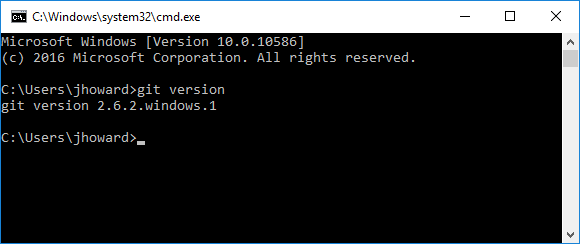




Note this screen. I select “Use Git from the Windows Command Prompt” to avoid the warning on the third option for “regular” command prompts, but have a specialized docker development command prompt which effectively changes it to “Use Git and optional Unix tools from the Windows Command Prompt”.

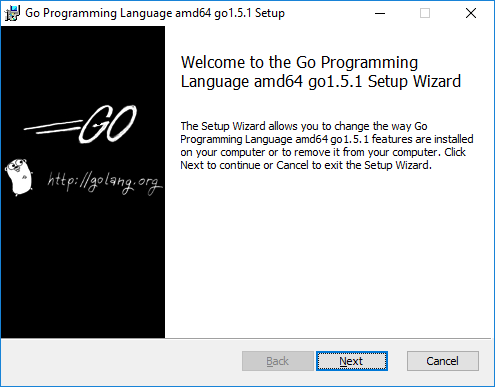


When it has finished installing, open a new command prompt and verify it has been installed correctly by running git version.



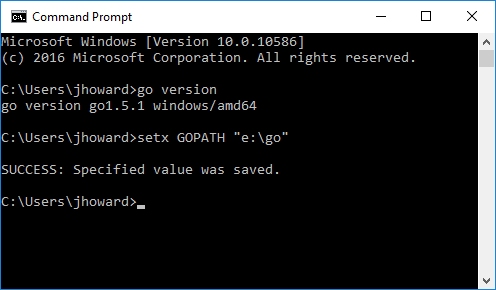
## GoLang

Select the matching version of golang to that being used by Docker. Currently this is 1.5.1. <https://golang.org/dl/>. Use the 64-bit installer with the default options.



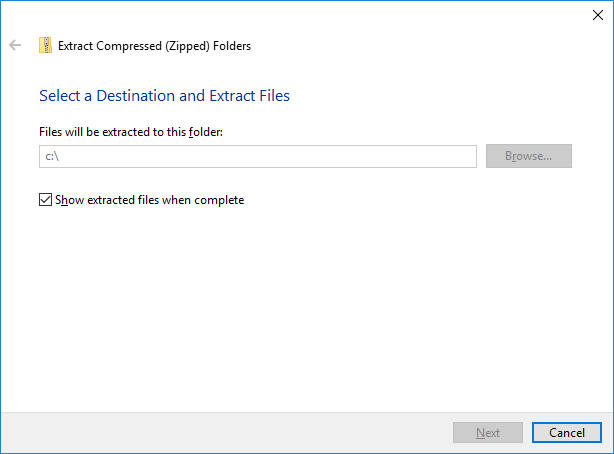
When it has finished, restart your machine (the installer doesn’t seem to set things up correctly until a reboot), open a new command prompt and verify it has been installed correctly by running go version.

Set your base GOPATH to where you will (in a few steps time) clone your sources. In my case, I use my E:\go drive as the root. setx GOPATH “e:\go”

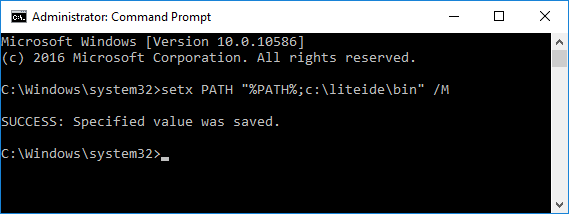


## LiteIDE

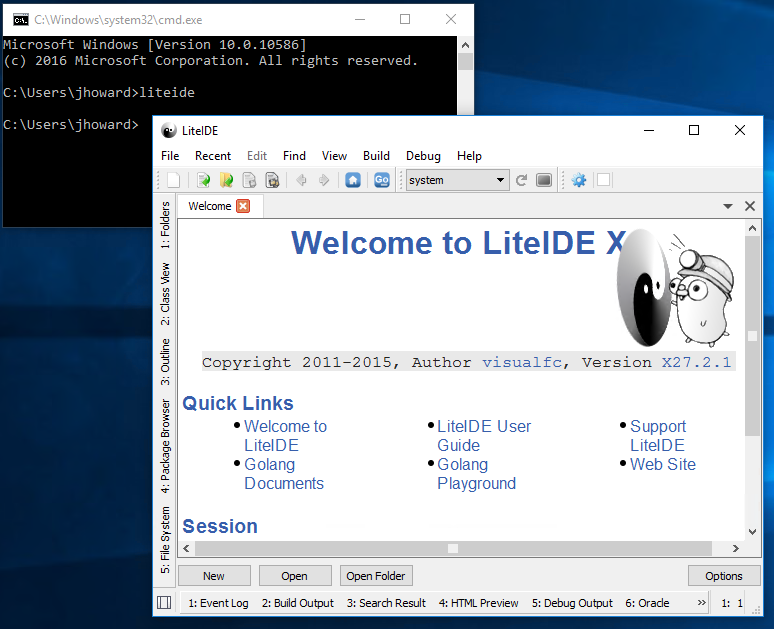
LiteIDE is a great tool for writing GO code, particularly as it formats (gofmt) code automatically for you, has syntax checking, intellisense and some (limited) debugging capability. Download the latest Windows 64-bit zip from <http://sourceforge.net/projects/liteide/files/> and unzip it to c:\. Note this is correct as the zip has a liteide directory in it, so the executable will end up under c:\liteide\bin.



Now update your PATH environment variable to include c:\liteide\bin by running the following command from an elevated command prompt setx PATH “%PATH%;c:\liteide\bin” /M



Now from a new command prompt (not elevated) run liteide, and verify it opens correctly.

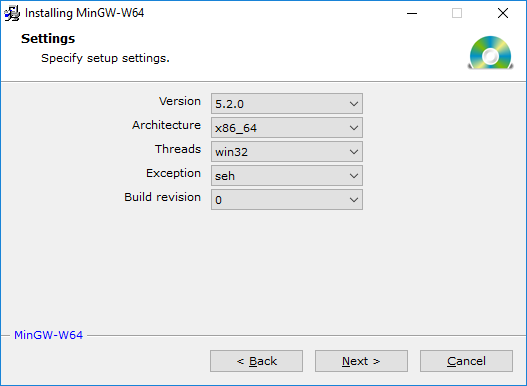


## MinGW-w64

You will need MinGW-w64 to compile CGO code. Download the latest installer from <http://sourceforge.net/projects/mingw-w64/>.



I use Architecture: x86\_64, Threads: win32 and Exception: she



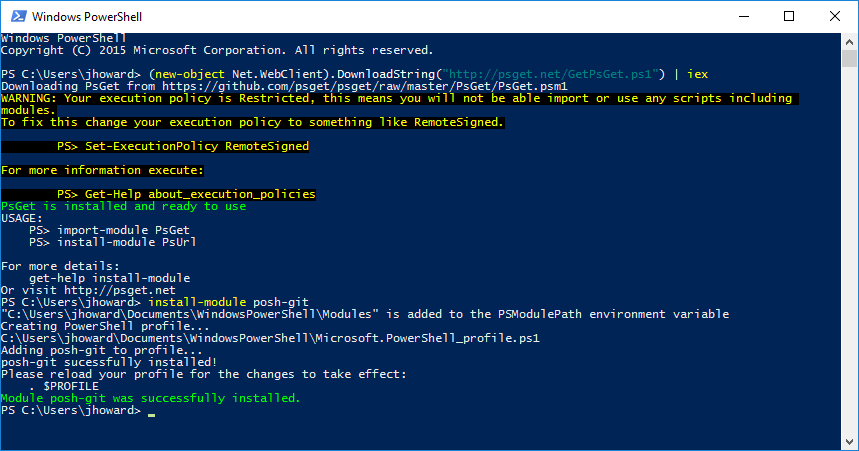
Keep a note of the path it installs in – you’ll need this later (but don’t worry if you lose it – we can find it again). In my case it’s C:\Program Files\mingw-w64\x86\_64-5.2.0-win32-seh-rt\_v4-rev0.

## Powershell Git (Posh-Git)

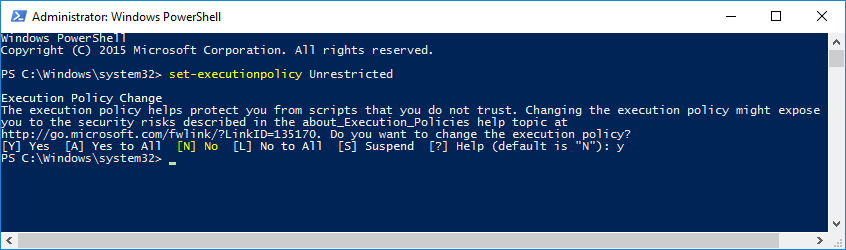
This is a tremendously helpful utility that gives useful hints on your command prompt such as the branch you are on, files counts changed, rebase assistance and so on. Totally invaluable!

From a PowerShell prompt, enter the following two commands:

(new-object Net.WebClient).DownloadString(“<http://psget.net/GetPsGet.ps1>”) | iex  
install-module posh-git

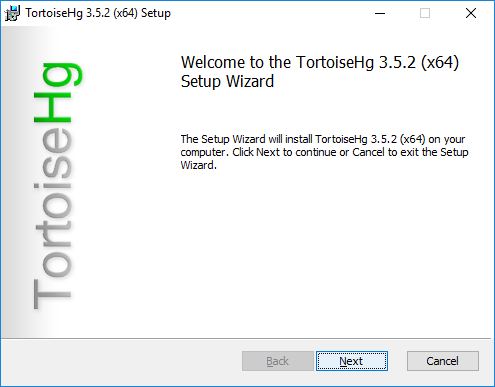


It may be useful to change your execution policy while you’re at it by running Set-ExecutionPolicy Unrestricted from an elevated PowerShell prompt. Be aware of the implications of this though.

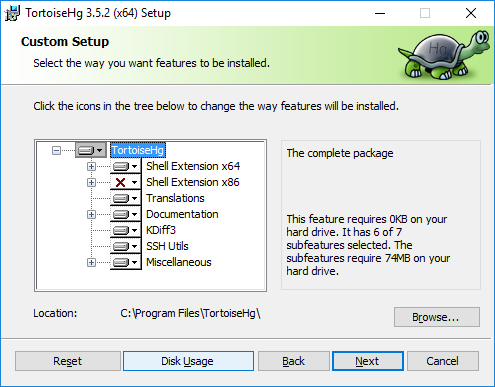


## TortoiseHg (Mercurial)

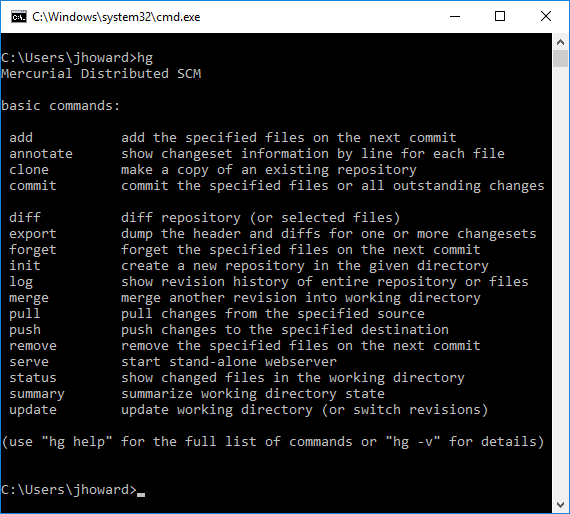
You may need this for cloning some sources (not docker core) using hg, or for vendoring. Run the latest TortoiseHg with Windows Explorer shell integration 64-bit installer from <https://www.mercurial-scm.org/wiki/Download#Windows>



I use the default installation options



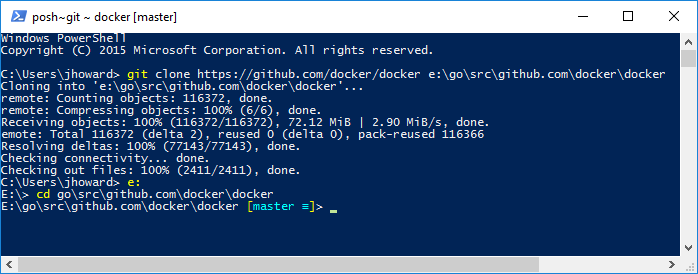
(Note I had to reboot for the installation to complete correctly). After a reboot, verify hg is installed from a command prompt.



## Clone the sources

Now you have the key software installed, clone the docker code. This will also allow you to verify that Posh-Git is installed correctly. I tend to use a separate drive for my sources, away from my operating system disk. In this case, it’s my E: drive.

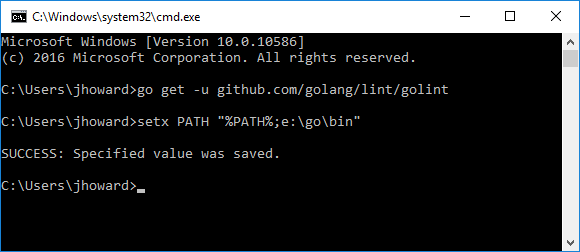
From a PowerShell prompt, run git clone <https://github.com/docker/docker> e:\go\src\github.com\docker\docker, then go to e:\go\src\github.com\docker\docker. All being well, you’ll notice “master” in your command prompt.



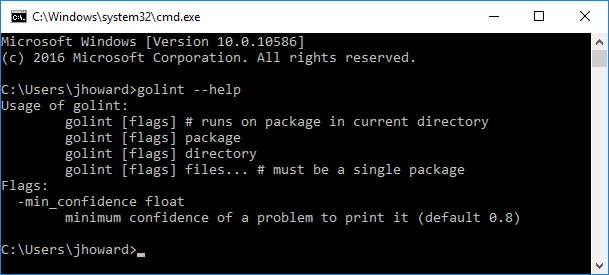
## Install some utilities

### Golint

From a command prompt: go get –u github.com/golang/lint/golint. This will install it to e:\go\bin. Now you need to update your path to include that path. Run setx PATH “%PATH%;e:\go\bin”

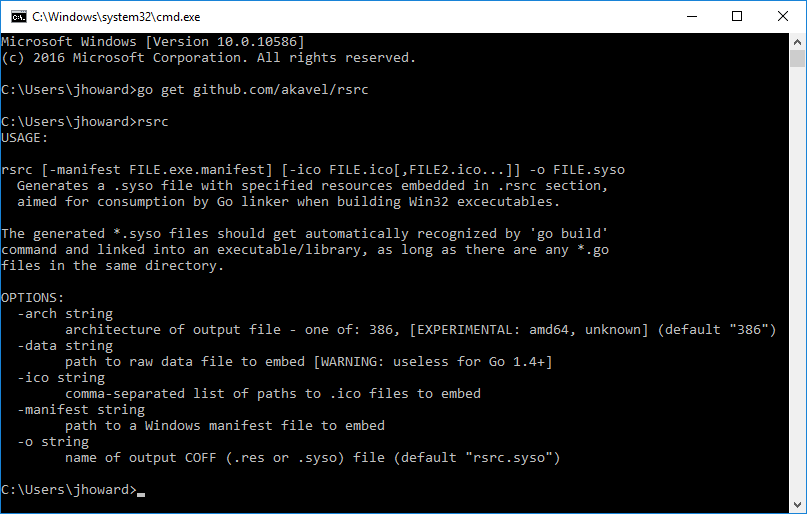


Open a new prompt (to pick up the path) and verify golint was installed correctly.



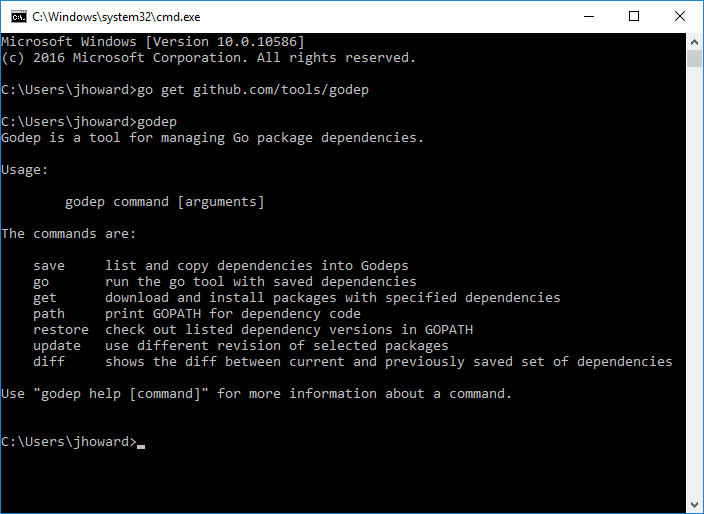
### Rsrc

This is a utility which allows manifesting of docker.exe, and embedding of the icon. Run go get github.com/akavel/rsrc and then rsrc to verify it has installed correctly.



### Godep

While not needed for docker core, if you need to work on some of the vendored packages, you’ll need Godep to build them. Run go get github.com/tools/godep and then godep to verify it has installed correctly.

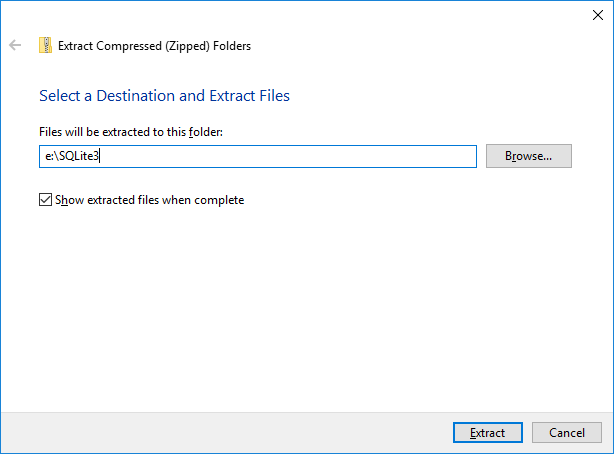


## Build SQLite3.dll

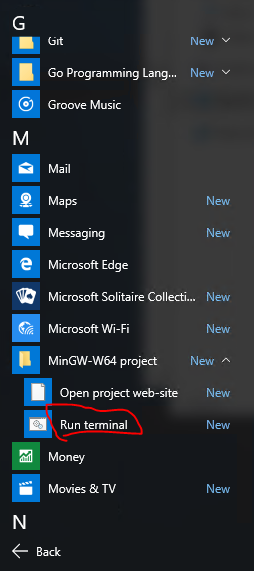
Docker has a dependency on SQLite for its graph (storage) database. There are three ways of building docker. The official way is a cross-compile from Linux. For development purposes on Windows, you have a couple of choices – to statically include SQLite in the binary, or dynamically link to SQLite. The latter is significantly faster to build (and what I use) as it doesn’t have to recompile all the SQLite sources. However, what it means is that you will need a 64-bit version of SQLite3.dll on each machine running docker.

When I originally started working on docker, there wasn’t a downloadable version of the 64-bit DLL for Windows available at <https://www.sqlite.org/download.html>. There is now, but for completeness, I still tend to build my own from amalgamation sources.

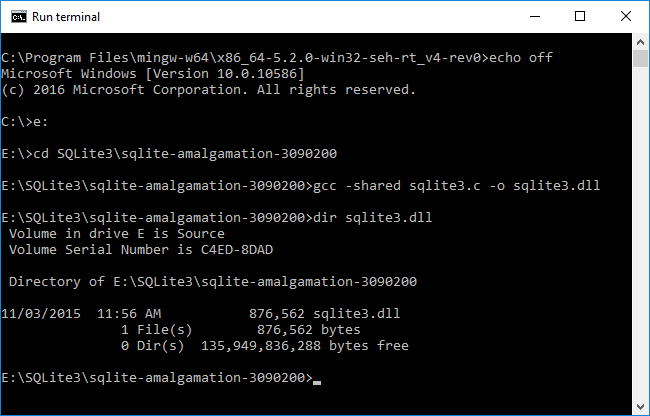
Download the latest amalgamation sources from <https://www.sqlite.org/download.html> and extract them to e:\SQLite3.



Open a MinGW-W64 command prompt from the start menu and change the directory to where your amalgamation sources are (eg e:\SQLite3\sqlite-amalgamation-3090200).



Enter the following command to build the dll: gcc –shared sqlite3.c –o sqlite3.dll, and verify the file exists.



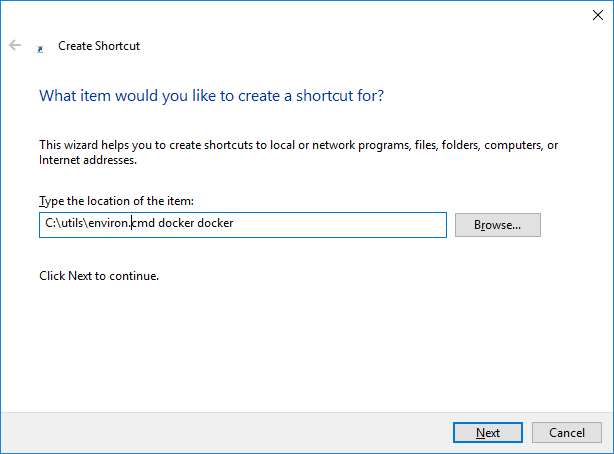
Copy the DLL to c:\windows\system32 on any machine you are going to be running the development docker binary from an elevated command prompt.

## Setup a custom development shell

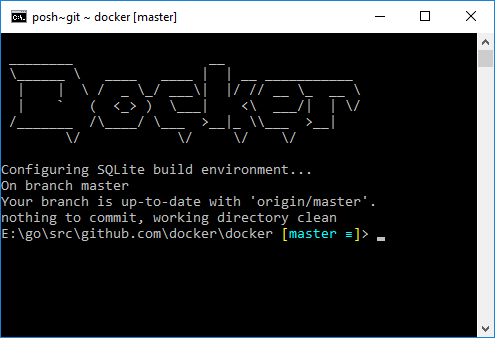
I have a few scripts that customize my environment depending on what I’m building – for example it could be docker/docker core itself, but it might be libcontainer from opencontainers/runc, or docker/libnetwork. Here’s a very simple script (I toned it down from what I actually have to make it easier to understand).

LINK TO SOMEWHERE ON GITHUB

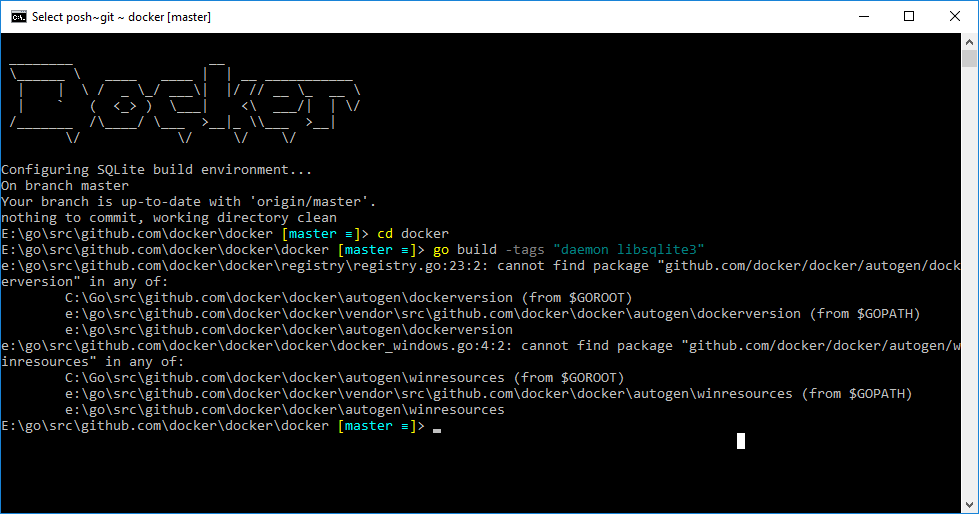
Create a desktop shortcut to “c:\utils\environ.cmd docker docker”



You just need to open your command prompt now from your shortcut



OK, let’s try a compile – we can see quite how really close we are getting, but there’s a tiny bit more still to do. From the docker\docker\docker (3 dockers) directory, run go build –tags “daemon libsqlite3”.



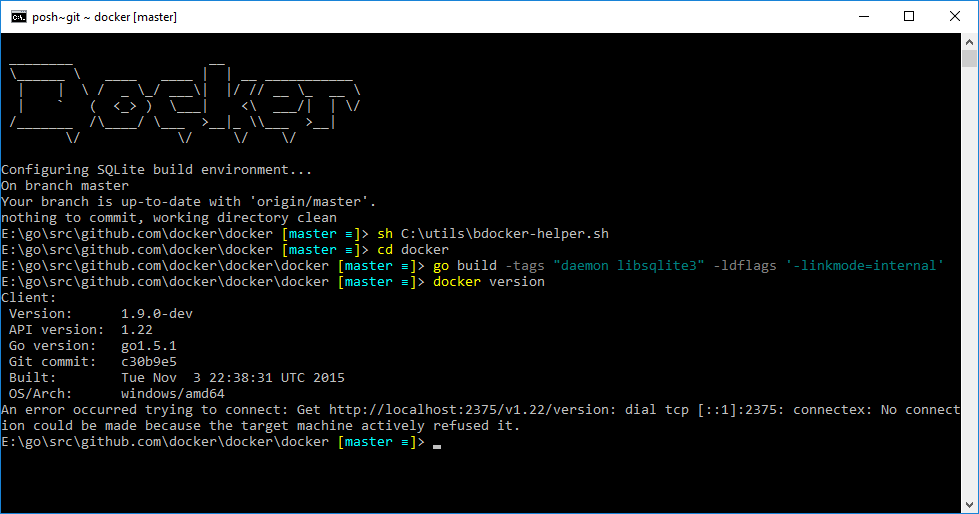
You’ll see it fails on autogen errors. To fix these, we’ll need another script.

## Fix autogen and our first successful compilation

Create the bdocker-helper.sh in your c:\utils directory with the following contents:

#!/usr/bin/env bash  
set –e  
VERSION=$(< ./VERSION)  
if command -v git &> /dev/null && git rev-parse &> /dev/null; then  
 GITCOMMIT=$(git rev-parse --short HEAD)  
 if [ -n "$(git status --porcelain --untracked-files=no)" ]; then  
 GITCOMMIT="$GITCOMMIT-dirty"  
 fi  
 BUILDTIME=$(date -u)  
elif [ "$DOCKER\_GITCOMMIT" ]; then  
 GITCOMMIT="$DOCKER\_GITCOMMIT"  
fi  
source hack/make/.go-autogen

Then run it as sh c:/utils/bdocker-helper.sh (note /, not \) from your github.com\docker\docker directory, before trying the build again, but this time with the full set of compile flags: From docker\docker\docker, go build –tags “daemon libsqlite3” – ldflags ‘-linkmode=internal’.



Success! Try out the newly built binary by running “docker version”. The error about not connecting is fine at this point – you don’t have a daemon running.

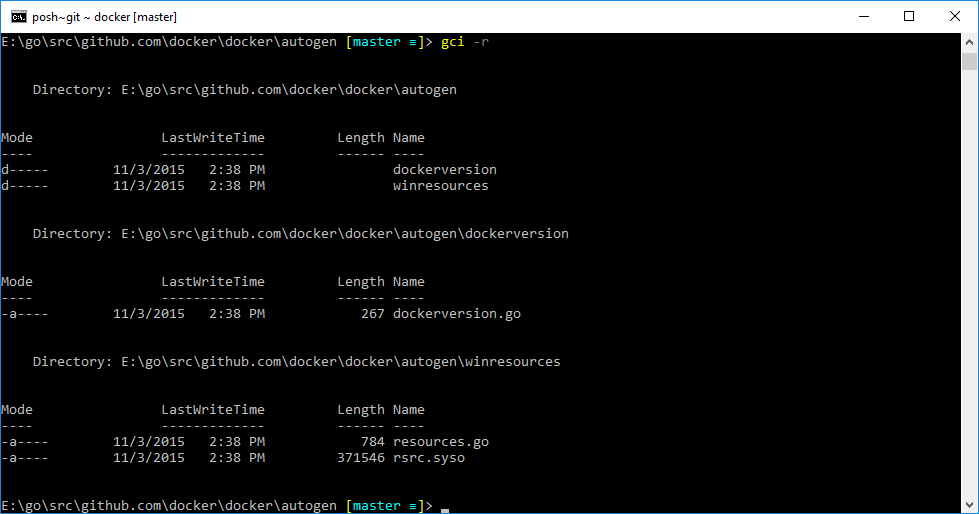
Tip: if docker version just returns immediately, are you sure you copied sqlite3.dll to your c:\windows\system32 directory?

## Explaining autogen and the compile flags

In the previous step, we built our first docker.exe binary, which can be used as both a daemon and a client. But some of the incantations in the build steps need some explanation. First the autogen script.

This sets up some environment variables in the same way that the official docker make file would, before calling .go-autogen in the docker sources itself.

.go-autogen creates the top-level directory ‘autogen’. For Windows, this contains two further sub-directories, dockerversion and winresources.



Dockerversion is used to embed the git commit, version and buildtime into the resulting binary.

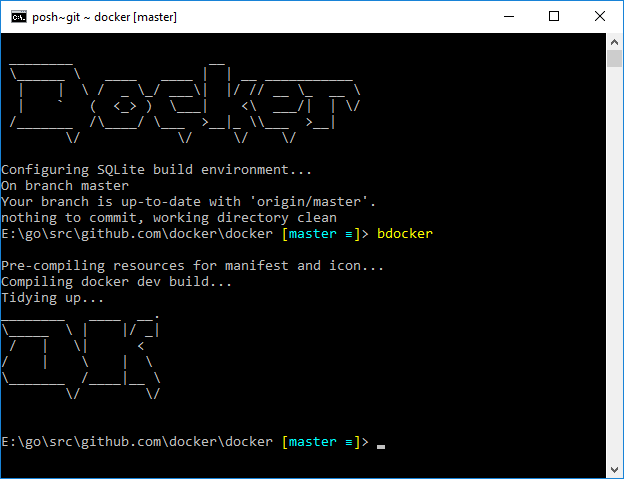
Winresources is used for manifesting the docker.exe and for embedding an icon. <https://msdn.microsoft.com/en-us/library/windows/desktop/dn481241(v=vs.85).aspx>

So moving on to the build flags incantations: go build –tags “daemon libsqlite3” – ldflags ‘-linkmode=internal’

* Build tag ‘daemon’ means build the binary including the daemon part of the sources
* Build tag ‘libsqlite3’ means we are dynamically linking to sqlite3.dll, which must be present at runtime. You can absolutely leave this tag out for static linking, but the compilation will take significantly longer (approximately 2.25x), but you won’t need sqlite3.dll.
* The ldflags are a workaround needed following the change from go 1.4.3. to 1.5.1. Here’s some more reading if you’re interested in why: <https://github.com/golang/go/issues/13070>

## Wrapper script for dynamic development build

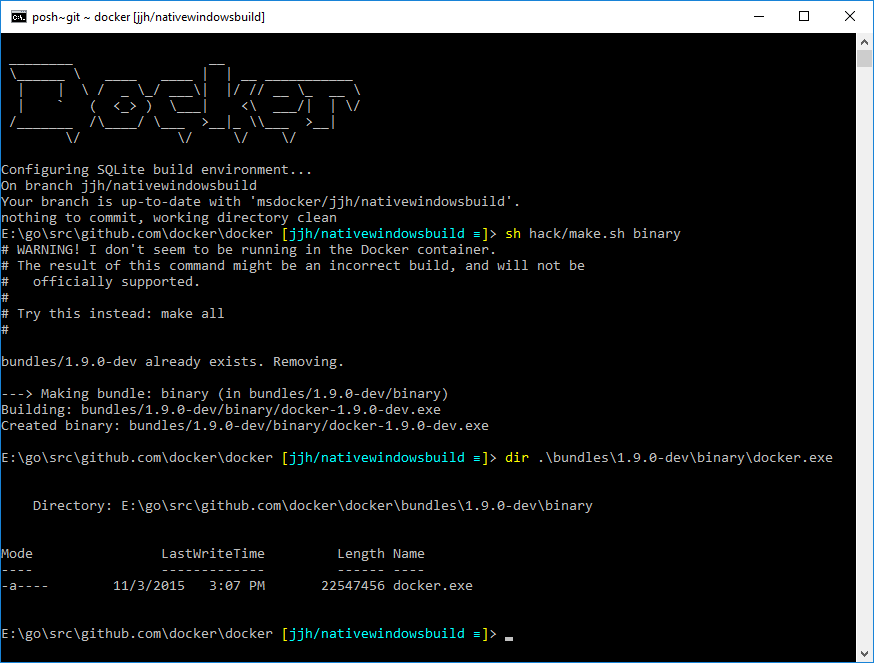
The simple wrapper script I have can be found here. It calls the helper-function for autogen, before calling go build. It’s called bdocker.cmd. Save it in your c:\utils directory.



## Using the standard build scripts

What we have done above is effective just a giant wrapper for the inbox scripts. While we can’t use the makefile natively on Windows (as it would attempt to build docker in docker which isn’t yet supported on Windows), you can leverage what it calls by running sh hack/make.sh from the github.com\docker\docker directory.

The final binary will be in bundles/version/binary/docker.exe, for example bundles/1.9.0-dev/binary/docker.exe



If you’re wondering why I’m not on the master branch, it’s due to this PR which wasn’t in the master docker code base at the time of writing: <https://github.com/docker/docker/pull/17544>

If you’re also wondering why the size difference between the official build scripts and the simpler static build called in bdocker.cmd, it’s simply that the official docker build adds a few more tags. Currently it runs

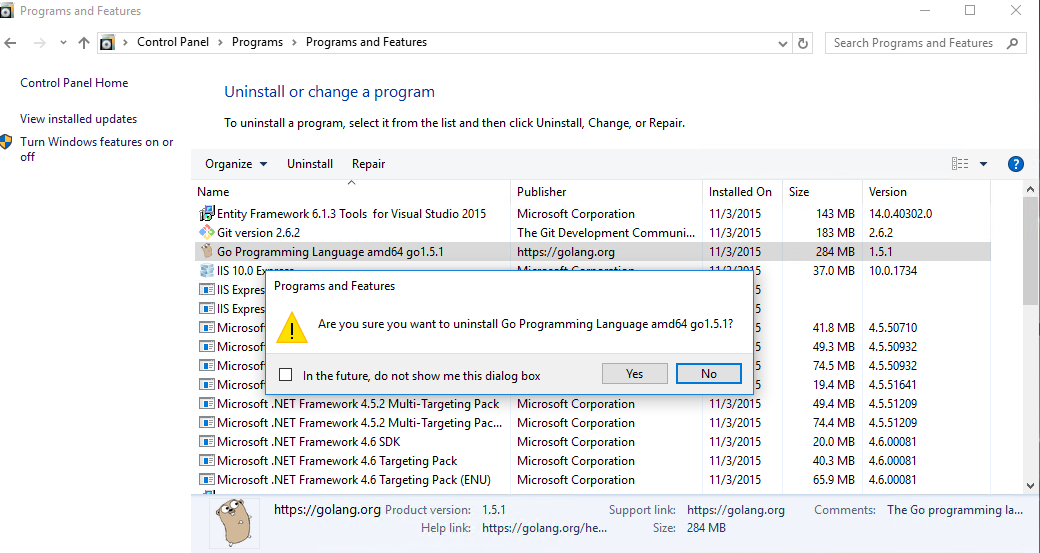
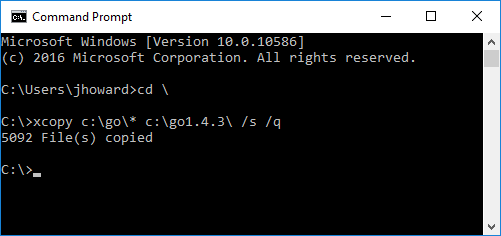
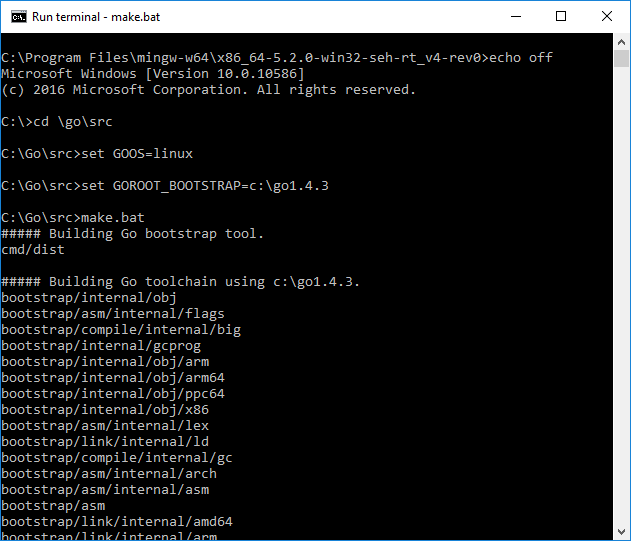
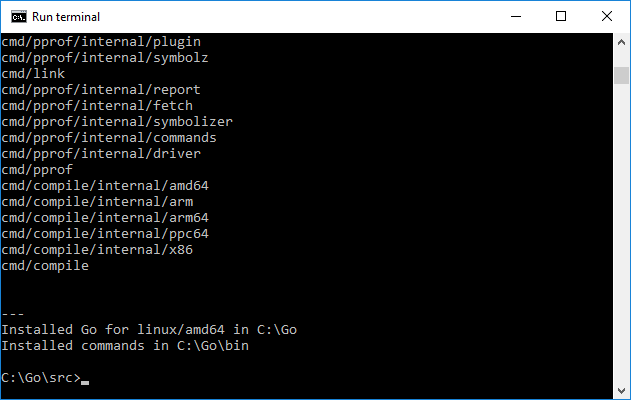
go build –a –tags netgo static\_build sqlite\_omit\_load\_extension daemon btrfs\_noversion libdm\_no\_deferred\_remove –installsuffix netgo –ldflags –w –extldflags “-static” –linkmode=internal

## Cross-compiling

Generally you won’t need to build a Linux, Darwin or FreeBSD binary from Windows, as you can just ‘make cross’ or ‘make binary’ from your Linux VM. However, it can be extremely useful to verify (some of) your changes still compile cross-platform as you change packages.

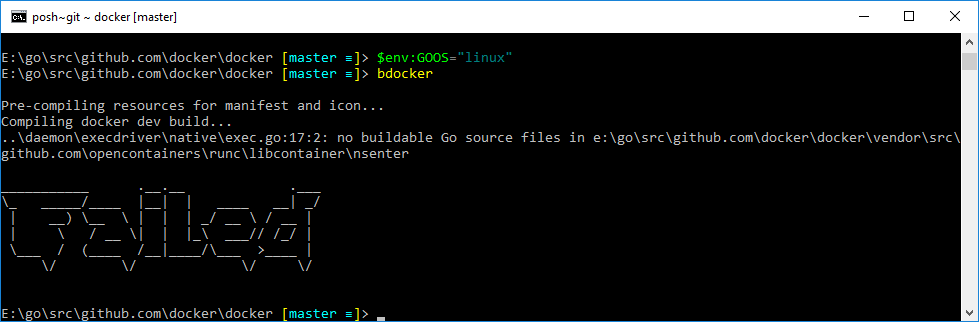
With go 1.5.1, there is an interesting new element which means that you can no longer just set your GOOS=linux and run c:\go\src\make.bat to compile the cross-platform tools. The tools are now written in go, so you need a previous version of go to bootstrap the cross-compile elements of go 1.5.1. It’s yet another version of the “what came first, the chicken or the egg” question.

Unfortunately also, at least when installing from MSI as I did above, you can’t have both go 1.4.3 and go 1.5.1 installed simultaneously on Windows. There is a reasonably simple workaround.

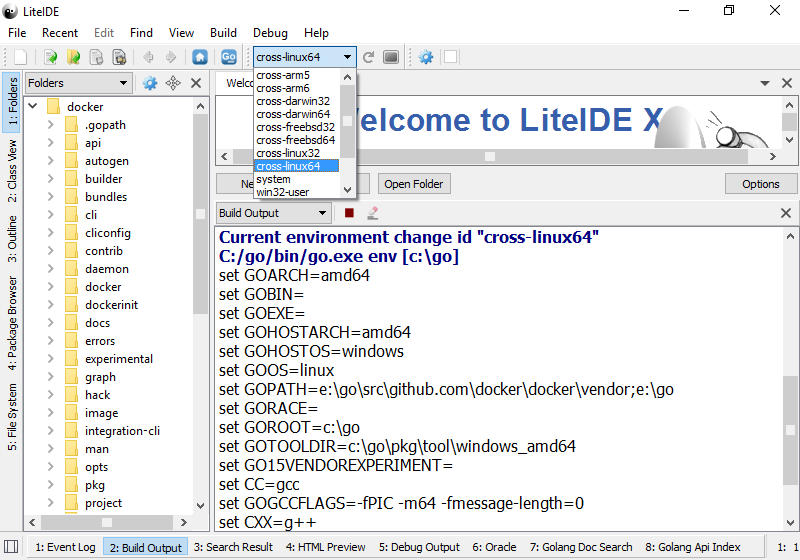
* Assuming you have go 1.5.1 installed, uninstall it from the control panel.  
  
* Using the link from earlier, install 64-bit go 1.4.3 to c:\go
* Make a copy of c:\go into c:\go1.4.3. Run xcopy c:\go\\* c:\go1.4.3\ /s /q  
  
* Uninstall go 1.4.3
* Re-install 64-bit go 1.5.1 into c:\go
* Start a mingw-64 terminal (as you need gcc for this step). Set your GOOS environment variable to your cross-platform target, such as ‘linux’; set your GOROOT\_BOOTSTRAP to the location of your xcopied go 1.4.3 install; and run make.bat  
    
    
  …  
  

You can find more information on bootstrapping at <https://docs.google.com/document/d/1OaatvGhEAq7VseQ9kkavxKNAfepWy2yhPUBs96FGV28/edit> and <http://dave.cheney.net/2015/03/03/cross-compilation-just-got-a-whole-lot-better-in-go-1-5>

So with that, now we can attempt cross-compile. One day I will figure out why overall build fails (something to do with cgo and how I’ve not quite got cross-compile platform environment quite correct), so this isn’t particularly useful in itself. Try it for yourself: set GOOS environment variable to “linux” and run bdocker.cmd

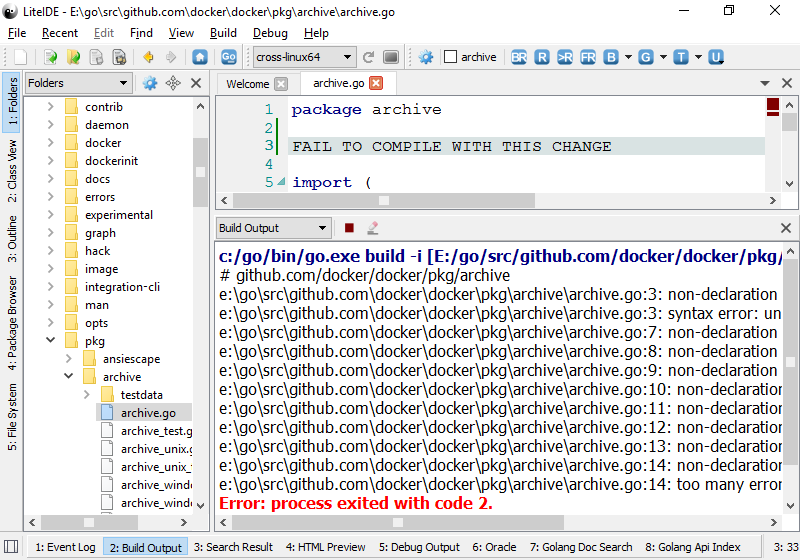


But where it is useful is for confirming compilation changes to unit tests or some packages (just not daemon, probably others). Fire up LiteIDE and add folder e:\go\src\github.com\docker\docker. In the drop-down on the toolbar, select cross-linux64

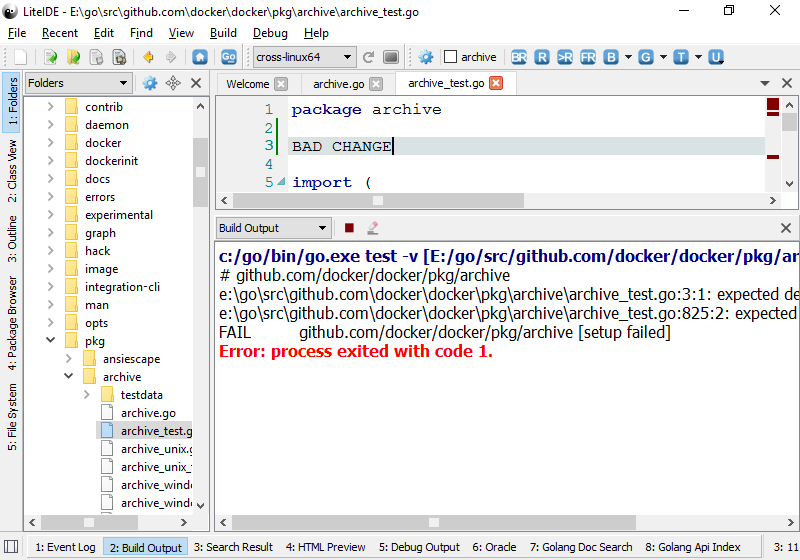


Note how in the Build Output tab, the environment has switched the GOOS to ‘linux’.

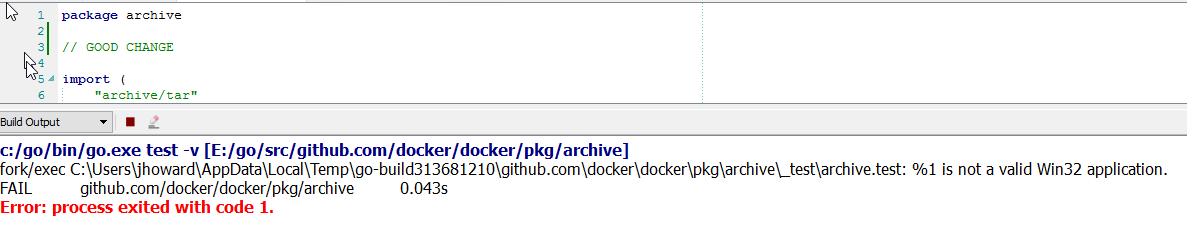
Let’s make a pretend change to core docker code that we want to confirm compiles on Linux. Navigate to pkg\archive and open archive.go. Put in a change that obviously won’t compile such as shown below. Then hit Control-B to kick off the compile:



Similarly, you can do the same with a \_test.go file such as pkg\archive\archive\_test.go, and use Control-T instead of Control-B to kick off unit testing. We can see compile errors very similarly:



However, if Control-B or Control-T succeed, that means that the change at least compiles, although can’t be run. The error you will see for test is something like that shown below – no compile errors, but the resultant binary won’t run on Windows as it’s not a Windows binary!



Of course the above are no substitute for actually building and testing Linux on Linux, but they are useful tools to have in your back-pocket as quick checks.

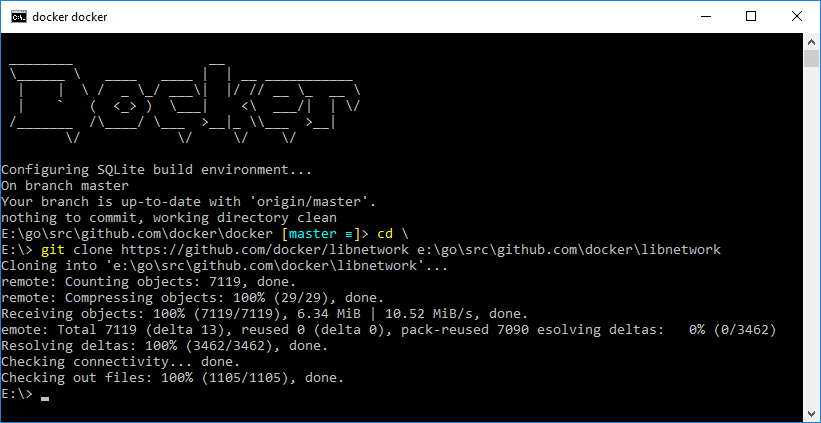
## Libnetwork

Libnetwork is an example of a “vendored” repo in docker. If you look at the source tree, you’ll notice there are a bunch of vendored repos under github.com\docker\docker\vendor\src\github.com\...

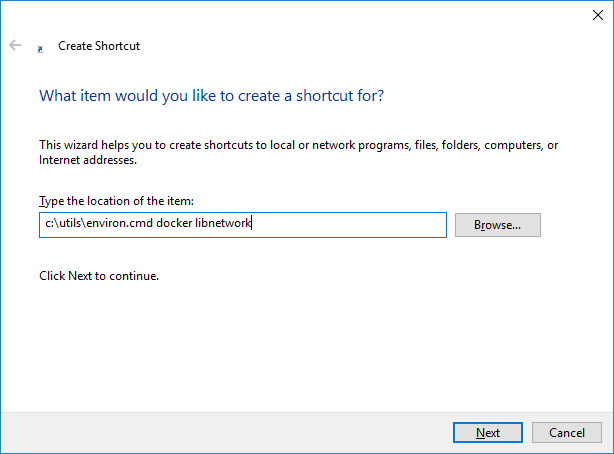
Libnetwork lives under vendor\src\github.com\docker\libnetwork.

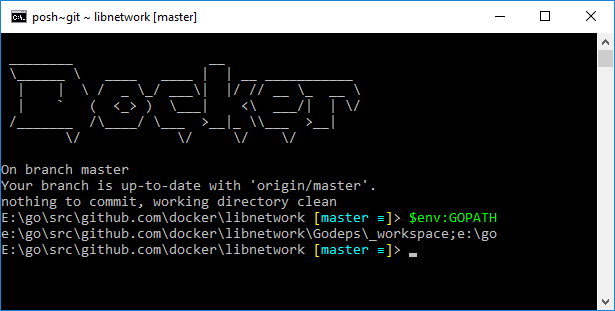
If you need to make a change to libnetwork for use in docker, or any other vendored repo for that matter, you should **not** make the change directly in the docker\docker path. Instead, you should clone the repo, submit a PR back to libnetwork, and then open a PR in docker which includes the re-vendored repo.

First, you need to clone libnetwork. Run git clone <https://github.com/docker/libnetwork> e:\go\src\github.com\docker\libnetwork.

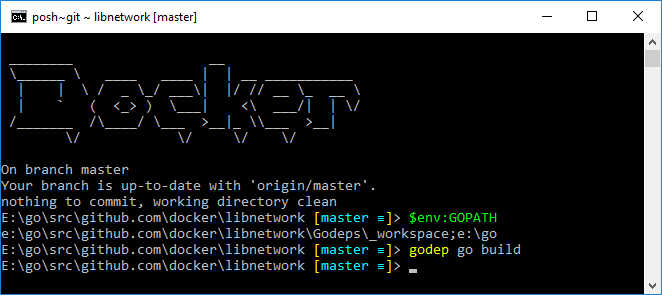


Then create a new shortcut for developing libnetwork (this simply sets the right gopath, which is slightly different to core docker). Using our environ.cmd script, create a shortcut to “e:\utils\environ.cmd docker libnetwork” and open it.



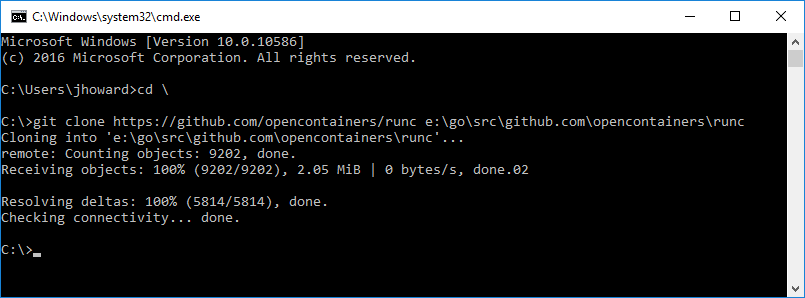


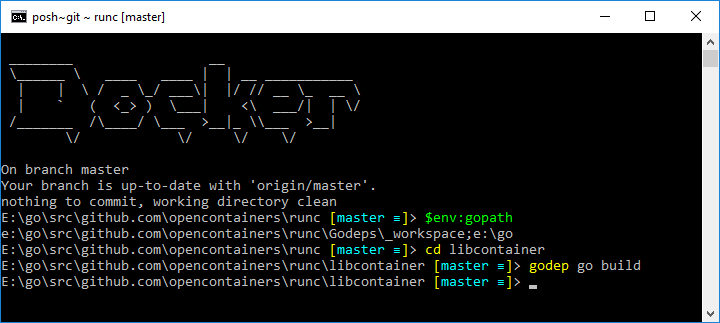
Notice that the GOPATH now points to a Godeps\\_workspace subdirectory. This is because libnetwork, and several other vendored packages, are built using go dep. To build, rather than “go build”, use godep go build.



Libcontainer

Libcontainer is similar to libnetwork above, but can be found as part of the opencontainers/runc project. To clone, run git clone <https://github.com/opencontainers/runc> e:\go\src\github.com\opencontainers\runc. You shortcut this time should be c:\utils\environ.cmd opencontainers runc.



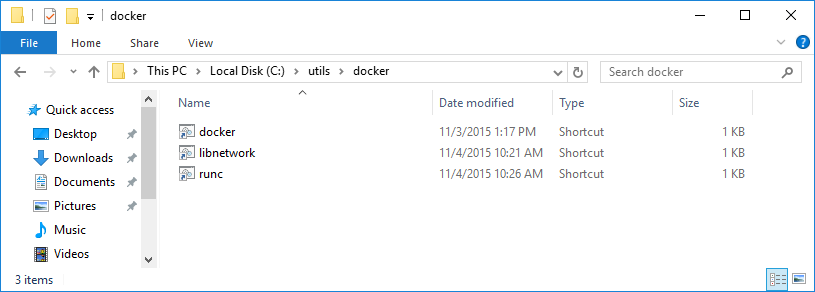


Vendoring

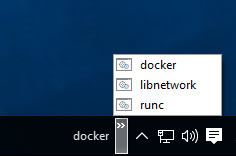
## Shortcuts tip



At this point, if you’ve followed everything, you probably have a few shortcuts on your desktop to open command prompts for developing in various repos. I find it particularly useful to move these shortcuts to a “docker” folder (eg c:\utils\docker) and create a taskbar toolbar.



Now right-click on the taskbar, and select “Toolbars/New toolbar…”, pointing to your c:\utils\docker folder.



So now you have a very convenient way of launching a development command prompt.

I hope that was hopeful to some of you!

Cheers,  
John.