Cross compiling for Debian Wheezy on the Beaglebone Black using a Ubuntu chroot

CMPT 477
Group 7

The steps outlined in this guide are for cross compiling for a Beaglebone black running Debian Wheezy from a Ubuntu 14.04 host. For reasons outlined below, this should only be necessary where the project has armhf dependencies and therefore the arm-linux-gnueabi-gcc can not be used. However it should be useful in any case where an older or different Debian cross compiler or toolchain is required, and is a faster solution than setting up a new VM due to the ease of use of the debootstrap tool.

Motivation

There are two arm cross compiling toolchains available in Ubuntu 14.04, arm-linux-gnueabi-gcc and arm-linux-gnueabihf-gcc, targeting the armel and armhf architectures respectively. Both can be used to compile for the Beaglebone Black because armhf is backwards compatible with armel binaries (but not vice versa). For projects depending on libraries built for armhf however, the armel toolchain will compile but the resulting binary will not be able to link to armhf libraries at runtime. The solution is to use the armhf toolchain, but unfortunately the version packaged for Ubuntu 14.04 uses a newer version of glibc (2.17) than the armel version and that version is too new to be compatible with Debian Wheezy on the Beaglebone Black.

If armhf features are not important you could also find or compile versions of those libraries for armel, but in our case we have an app that is heavily dependent on floating point math, including alsa, libsndfile and realtime audio filtering, so we decided that using armhf is the better solution. Also, by setting up a chroot build environment, it allows compilation on different distributions and hopefully any future Debian based system.

Another way of solving this problem would be to link an older glibc or statically linked a newer version but this is not a recommended practice and for this reason gcc does not make it easy (we tried). On linux, unlike windows, when compiling for older targets the recommended practice is to compile on an older system.

We could also have updated libc or Debian on the board however we wanted to maintain compatibility with this specific Debian version, since it is the official version for Beaglebone and used by all the other boards.

Alternatively, another potentially good solution but which might be more work, and also compile slower, would be to set up a virtual machine with QEMU, which is capable of emulating the arm cpu. This solution would allow you to compile within an armhf Debian Wheezy system, or the exact system of the target, and you would not have to manually copy the dependencies needed, simply apt-get the dependencies for armhf directly, and avoid dependency on any external cross compiler toolchain.

Note:

It is possible and in fact much easier to to not cross compile, and compile directly on the Beaglebone black (or any system that runs Debian) so this guide should be taken as an educational exercise and not as strictly necessary. It would probably be much more helpful for a much lower powered system than the beaglebone black or for large projects where compilation time is a factor. Also this guide assumes some familiarity with makefiles and linux in general and may not describe every step in detail.

Steps:

The first step is to get the necessary headers and shared libraries for each armhf dependency for the target. Because you will have to install them on the beaglebone black anyway you can simply apt-get install them there and copy them to your host.
For example, for the alsa library, libasound, to locate the library:

```
# ldconfig -p | grep libasound

     libasound.so.2 (libc6,hard-float) => /usr/lib/arm-linux-gnueabihf/libasound.so.2
```

to locate the headers (note: locate may have to be installed on the BBB, and requires you to run the command updatedb first)

```
# locate asoundlib.h

     /usr/include/alsa/asoundlib.h
```

Copy /usr/lib/arm-linux-gnueabihf/libasound.so.2 and /usr/include/alsa from the target to /home/myproject/cross-deps/include and cross-deps/lib on the host. Use nfs, scp or something similar to copy from target to host.

Add the necessary flags to your makefile, ours looks similar to this

```
CC_TARGET=arm-linux-gnueabihf-gcc
CFLAGS_TARGET=-Icross-dep/include
LDFLAGS_TARGET=-g -lm -Lcross-dep/lib -Wl,-rpath=cross-dep/lib
-Wl,-rpath-link=cross-dep/lib -lsndfile -lasound -ljson -lrt
```

-Icross-dep/include tells GCC to look in this folder for the header files for your dependency
-Wl, is used by gcc to send flags to the linker, ld, in this case the cross-dep/lib directory, to link the armhf shared libraries we copied over.

You are now ready to set up the chroot environment. We will install the new system to a directory inside the project we are trying to compile

```
# apt-get install deboostrap
# cd /home/myproject
# mkdir precise-chroot
# sudo debootstrap --variant=minbase precise ./precise-chroot
http://archive.ubuntu.com/ubuntu/
```

This downloads Ubuntu 12.04 (precise) from the ubuntu archive URL into the folder ./precise-chroot. The variant flag can specify different packages to attach, but since we will manually install the packages we need, you can use minbase, which is the smallest one.

Now we need to make our project directory accessible inside the chroot, however it is only possible to create a symlink into, but not out of a chroot system, so we use mount --bind.

```
# sudo mkdir -p precise-chroot/home/myproject
# sudo mount --bind /home/myproject ./precise-chroot/home/myproject
```

Now we can enter the chroot with

```
# sudo chroot precise-chroot
```

This changes precise-chroot into the apparent root directory, now all commands we run will use the Ubuntu 12.04 versions. Our userspace OS has been replaced while running the same kernel as before. Inside this system we need to install make and gcc-arm-linux-gnueabihf. gcc-arm-linux-gnueabihf is not available from the main ubuntu repositories in this system, it is instead available in the precise ‘universe’ repository, which is for community maintained software. We can add it with

```
# echo deb http://archive.ubuntu.com/ubuntu precise universe >>
/etc/apt/sources.list"
```

Now we can apt-get update and download what we need

```
# apt-get update
# apt-get install make
# apt-get install gcc-arm-linux-gnueabihf
```

Because our projects dependencies are for arm they are in the cross-dep folder and we don’t need to install them here. Now you should be able to build your project, as long as your make file is setup correctly. In our case, we had seperate builds on the host and the target and specified the target build with the suffix .target.

```
# cd /home/myproject
# make myproject.target
```

To run the build process in one step, you might want to create two shell scripts, one that will setup the chroot environment, and another that enters the chroot and runs make.

One important note is that running chroot from inside a shell script will interrupt execution of the shell script until you leave the chroot, so the solution is either to use here documents or pass the program to be run to chroot in a single command.
Here documents are typically used to pass a block of text to stdin for an interactive program. For example, in your script to set up the environment, add

```
sudo chroot precise-chroot << CUT
apt-get update
apt-get install -y make
apt-get install -y gcc-arm-linux-gnueabihf
CUT
```

or you can simply pass one program to be run inside the chroot, such as bash, with parameters (another script). Here is an example build script

```
sudo chroot precise-chroot /bin/bash -c "cd /home/myproject/; make
bin/myproject.target"
```

You now have a simple way to cross compile armhf binaries for Debian on the Beaglebone Black from any Ubuntu or Debian based host.