Cross Compilation Framework

18–342 Fundamentals of Embedded Systems

October 6, 2015

1 Introduction

After Lab 2 (part 1) you will be running your code on U-Boot directly. Therefore developing your code on the Linux environment on your boards/emulator and resetting the board so that you can run your code on U-Boot can be tedious and time consuming. In this handout, we will show how you can set up a cross compiler tool chain on your laptop so that you can develop, compile your code on your laptop and then transfer it to the emulator to run it (or emulate running it) on the hardware board. This tool chain and environment will allow you to store, develop and maintain your code on any linux machine (if you cannot access any other linux machine, you should be able to use `unix.andrew.cmu.edu`) and use the same to compile your code. You will still develop and compile the code on the local machine and then load the binaries onto the gumstix.

You will necessarily have to use the cross-compiler tool chain for labs 3 and 4. The C libraries on verdex-pro were compiled to use a hardware FPU for all floating point computations (the unfortunate part is that the PXA270 does not have an FPU, the libraries should have been compiled to use software libraries for floating point computations). So if you are using any floating point operation (like a divide operation) your code will not compile using the gumstix compiler.

One of the biggest advantage of the cross-compiler tool chain is that you can use your favorite editor (no longer stuck with `vi`) to do the development and also the make process is quite fast on the cross-compiler tool chain. Moreover you won't have to wait for 10 minutes for your verdex-pro board to reboot in order for you to execute your application from U-Boot or develop your application in linux.

This is a guide on how to get this infrastructure working for you.

2 Cross Compiler Setup

To set up your workspace to correctly use the cross-compilation toolchain, please follow these steps:

1. Download the toolchain `opt.tar.gz` from Blackboard/Resources to your linux development machine.
2. If you don't have a linux development machine, log in to `unix.andrew.cmu.edu` using your andrew account and add `/afs/andrew.cmu.edu/usr6/rgandhi/gumstix/opt/bin` to your path. On bash or derivative shells (execute `/bin/bash` if the default shell is not `bash`), this can be performed with:

   ```
   export PATH=/afs/andrew.cmu.edu/usr6/rgandhi/gumstix/opt/bin:$PATH
   ```

   You may also add this to your `.bashrc` or your `.bash_profile`.

3. If you do have a linux development machine, un-tar `opt.tar.gz` in some folder (say the full path of the folder is `/cannonical/path/to/folder`). Set your path variable to include the tool chain. On bash or derivative shells, this can be performed with:

   ```
   export PATH=/canonical/path/to/folder/opt/bin:$PATH
   ```

   You may also add this to your `.bashrc` or your `.bash_profile`.

4. Test the installation and the path — try `arm-linux-gcc -v`. If this command is successful (you see a message saying "`Using built-in specs. ....`") , you know that you have set up the path to the cross-compilation toolchain correctly. The toolchain contains standard gcc and binutils programs such as gcc, ld, as, ar, objdump and strip.
   Instead, if you get an error saying "`/canonical/path/to/folder/opt/bin/arm-linux-gcc: No such file or directory`" even though you can find the compiler in the directory, you may be running on a 64-bit Linux system. You can verify this by typing `uname -i` on the terminal – if the result is `x86_64`, then you are on the 64 bit version. In this case you will need to install an additional package before you can use the cross-compiler. If you are on the 64-bit Linux system, install `libc6-i386` using the following command

   ```
   sudo apt-get install libc6-i386
   ```

   One you have installed the additional package, you should receive the correct output when you run `arm-linux-gcc -v`.

5. Edit all the Makefiles of your project directory to use the cross-compilation toolchain i.e. use `CC = arm-linux-gcc`, `AR = arm-linux-ar`, `OBJCOPY = arm-linux-objcopy`, ... Take one of your existing projects and compile it using:

   ```
   make clobber && make
   ```

You now have a working cross-compilation environment on your/andrew machine. You may use this to develop and compile programs.

3 Copying Executables to microSD Card in QEMU

You now need to setup your local machine to be able to download the binaries that you generate to the `sdcard.img` file for use with QEMU. You may want to create a backup copy of your original `sdcard.img` because the following steps will overwrite the contents of the original `sdcard.img` (in case something goes wrong).

1. Download the `bash` script `copy2sd` **in the directory** that contains `sdcard.img`.
2. Type the following command to ensure `copy2sd` is recognized as an executable

   ```
   chmod u+x copy2sd
   ```

3. Copy all the files you want to transfer to the microSD card by using

   ```
   copy2sd /path/to/firstfile  /path/to/secondfile . . .  /path/to/lastfile
   ```

4. Use the new `sdcard.img` in the QEMU command line and start QEMU.
5. After you type `mmcinit` on U-Boot, `fatls mmc 0` should show you all the files you had copied onto the microSD card. You can now use `fatload mmc 0` command to load these binary files to memory.

4 Using gdb with QEMU

One of the advantages of using QEMU to emulate the hardware board is the ability to use gdb running on your local (host) machine to debug binaries running on the emulator (target). In order to use gdb, you will need to invoke QEMU with the -s flag as shown below

```
qemu-system-arm -M verdex -pflash ./flash.img -nographic -sd sdcard.img -s
```

Press a key so that you can prevent U-Boot from loading the Linux image on the microSD card. The -s flag instructs QEMU to create a gdb server on the target that you can connect to from your host machine.

However, before we can start debugging our ARM kernel-code, we will need to install a version of GDB which runs on our x86/x86-64 based host machine but can debug the ARM-based Gumstix (a cross-debugger) boards. This cross-debugger can be installed on your host machine by using the following command:

```
sudo apt-get install gdb-multiarch
```

We can now start a gdb client on the host machine by entering the following command

```
gdb-multiarch
```

We can now set the appropriate architecture for the gdb running on the host machine and connect to the server running on the emulator by using the following commands:

```
gdb>set architecture armv5te
gdb>target remote localhost:1234
```

At this point, QEMU should cease emulation because its being controlled by gdb (it will stay stopped until you type continue in gdb).

We can also leverage symbolic debugging within gdb with just a few more commands (so you can type break main instead of break 0xa292a3ec). To support this, you first need to (re-)compile your code with debugging symbols enabled by adding the -g flag to the CFLAGS variable in your Makefile. Then, in gdb:

```
gdb> add-symbol-file path/to/your/kernel 0xa3000000
gdb> add-symbol-file path/to/your/user/binary 0xa000000
```

Congratulations, you can now inspect registers, memory, stack, and poke around symbolically within the comfort of a gdb environment!