



Prac 04: QEMU Arm Emulator and Kernel Building

[40 marks]

Contents

INTRODUCTION	1
OBJECTIVES.....	2
PART A: EXPLORING QEMU AND USING ANDRIOD.....	2
PART B: BUILDING THE LINUX KERNEL FOR ARM AND RUNNING IN QEMU.....	3
1. COMPILE QEMU FOR THE ARM.....	3
2. RUNNING THE U-BOOT (5 MARKS).....	3
3. USING THE “BOOTM” COMMAND TO BOOT AN ARM PROGRAM (10 MARKS)	3
4. SQLITE AS AN EMBEDDED DBMS (20).....	5

Introduction

Complex digital systems frequently comprise processors as well as dedicated digital and analogue circuitry. The processor may likely be in the form of an embedded microprocessor onto which dedicated embedded software is to run. But, as you may know from the Embedded Systems EEE3074W course, the selection process for choosing the processor can be a major factor in the success of the project. Furthermore, the development resources for the embedded processor (e.g., libraries and supported operating system) might – or might not – be adequate for the task. Processor emulators can be a significant cost saving – offering a means to essentially test a large part of a proposed development without any additional hardware (i.e., an approach like download your development platform and tools all from the comfort of your desk, laboratory or sundeck recliner depending where you would prefer to experiment with tools).

You can right away **jump to Part A on Page 2** if you don’t want to read the rest of this intro!

Students doing this practical are expected to have some prior experience with embedded systems development, the ARM processor and running embedded software on evaluation boards. This laboratory practical uses an ARM processor emulator together with building a Linux kernel to run on this virtual processor. This prac has specifically been included in response to embedded industries, both locally and internationally, having identified the need for computer engineering graduates to have experience in building Linux kernels and using emulators¹. This practical effectively combines the two.

1 These points are further substantiated by a host of literature articles, such as:
K. G. Ricks, D. J. Jackson, and W. A. Stapleton, “Incorporating embedded programming skills into an ECE curriculum,” ACM SIGBED Review, vol. 4, pp. 17-26, 2007.
G. Shen, “An Embedded System Curriculum for Undergraduate Software Engineering Program,” Software Engineering Research, Management and Applications, pp. 219-232, 2008.

The purpose of this lab is to familiarize the EEE4084F students with the process of developing with embedded Linux in the cross-compilation environment (i.e., the use of ARM is largely coincidental, much of the procedures learned here are generally applicable to use of embedded Linux). In this practical you will compile the kernel, libraries and applications and downloading the resulting images onto the virtual embedded target. We will use the QEMU emulator to run the embedded Linux applications (QEMU homepage is at: http://wiki.qemu.org/Main_Page). The plan is to run an ARM application program and boot the ARM Linux kernel. This emulation platform is similar to the VersatilePB platform (<http://www.arm.com/products/tools/development-boards/versatile/index.php>).

Objectives

There are two parts of this practical: Part A involves exploring QEMU, and Part B concerns booting and compiling the ARM embedded application using universal boot loader, u-boot, as well as exploring SQLite as an embedded systems DBMS.

The learning objectives of this lab are:

- Explore the QEMU, an embedded systems emulator and virtualizer environment
- Cross-compile and run a test program for the ARM architecture
- Understand how to configure and cross-compile an ARM embedded system application on a PC
- Boot the resulting test system image in QEMU
- Create an SQLite sample database (one of the favorite databases used for embedded systems) and perform an SQL transaction using it.

Beyond giving the student good depth of understanding practicalities of high performance embedded computing, this practical helps to familiarize students with using ARM processors that are commonly found in products such as network switches, mobile phones, and GPS navigators. It also introduce students in the development of embedded systems applications and emulation to verify before shipping to the fabric.

Part A: Exploring QEMU and using Android

Follow the steps below:

1. Install QEMU on Ubuntu using the following command (Only if you are running on your own PC:):

```
# sudo apt-get install qemu qemu-kvm-extras
```

(Since we are not allowed to install packages in the Blue refer to part B.1 for compiling Qemu from source package included in this prac folder – the Qemu binaries have already be installed on the machines; if it isn't on your machines please ask a tutor to help.)

2. Run the android image by executing this command on the directory where the Android froyo image resides:

```
$ qemu -hda lx_froyo.iso -L /usr/share/qemu -m 512
```

Part B: Building the Linux kernel for ARM and Running in QEMU

1. Compile QEMU For the ARM

```
# cd qemu-0.14.0
# ./configure --target-list=arm-softmmu
# make
# sudo make install
```

Setting PATH for ARM binaries

```
#PATH=~/.Prac03/arm-2011.03/bin:$PATH
```

2. Running the U-Boot (5 marks)

The universal bootloader, u-boot, is a crucial piece of software that runs on embedded platforms. Put in place and boot the linux kernel from a hard drive, a flash memory, network or serial line. In this part of Prac03, we will cross-compile uboot and use it to run an application compiled for ARM.

Go inside the created directory and run:

```
# sudo apt-get install uboot-mkimage (install mkimage)
# cd u-boot-2010.3 (Goto u-boot linux directory)
```

Configure to cross compile for the versatilePB Board by executing the following command:

```
# make versatilepb_config ARCH=arm CROSS_COMPILE=arm-none-eabi-
```

Now compile the linux kernel by executing the command below:

```
# make all ARCH=arm CROSS_COMPILE=arm-none-eabi-
```

This will start build the kernel using the correct ARM compiler. The compilation will create a u-boot.bin binary image in the folder ~/u-boot-2010.03. To simulate, change to ../qemu-0.14.0/arm-softmmu/ and run the following (note it should all be on one line):

```
# ./qemu-system-arm -M versatilepb -m 128M -nographic \
-kernel ../../u-boot-2010.03/u-boot.bin
```

Observe the outcome and Printout the Screenshot for the report. For gnu-linux operating systems, there are two mostly used commands to get list of commands to be used in the gnu-shell/command prompt. Use these commands to display the list of commands that can be used on the u-boot prompt. (5 marks)

NB: Press **"Ctrl + a"** and then press **"x" to exit QEMU** .

3. Using the "bootm" command to boot an ARM program (10 marks)

To display a string or messages in the VersatilePB board, we dont use straight forward c functions like printf(). We use the address where the UART0 is mapped: 0x101f1000. There is a register (UARTDR) that is used to transmit (when writing in the register) and receive (when reading) bytes. This register is placed at offset 0x0. We read and write at the beginning of the memory allocated for the UART0. Look at the test.c example code in the function print_uart0().

- The `volatile` keyword is necessary to instruct the compiler that the memory pointed by `UART0DR` can change or has effects independently of the program.
- The `unsigned int` type enforces 32-bits read and write access.
- The QEMU model of the PL011 serial port ignores the transmit FIFO capabilities; in a real system on chip the “Transmit FIFO Full” flag must be checked in the `UARTFR` register before writing on the `UARTDR` register.

The `test.c` program displays a simple “Hello World.” message, Compile the ARM program as explained below:

```
# arm-none-eabi-gcc -c -mcpu=arm926ej-s test.c -o test.o
```

Create a startup application bash program with contents such as in `startup.s` file in the Prac folder. One can replace `testfunc` word in the `startup.s` with the name of the function in the `test.c` program they want to run on the u-boot system.

```
# arm-none-eabi-as -mcpu=arm926ej-s -g startup.s -o startup.o
```

Another script you have to create is the linker helper, example is in the `test.ld` script that simply calls the startup binary to help the linker to add in the library to be created. It also indicates the memory size to be used in the test application. Now link the the binaries as follows:

```
# arm-none-eabi-ld -T test.ld test.o startup.o -o test.elf
```

Then create the binary as follows:

```
# arm-none-eabi-objcopy -O binary test.elf test.bin
```

At this point you can test the image of your test program to see if it works on its own without booting it from u-boot as follows:

```
# qemu-system-arm -M versatilepb -m 128M -nographic -kernel test.bin
```

Now create the U-Boot image `test.uimg` with:

```
# mkimage -A arm -C none -O linux -T kernel -d test.bin
-a 0x00100000 -e 0x00100000 test.uimg
```

With these options we affirm that the image is for ARM architecture, is not compressed, is meant to be loaded at address `0x100000` and the entry point is at the same address. We use “linux” as operating system and “kernel” as image type because in this way U-Boot cleans the environment before passing the control to our image: this means disabling interrupts, caches and MMU.

Now create a single binary simply with:

```
# cat u-boot.bin test.uimg > flash.bin
```

Run the binary image with:

```
# qemu-system-arm -M versatilepb -m 128M -nographic -kernel flash.bin
```

This will take us to the `versatilepb` prompt for our arm system we implemented.

Now we have to run the test image (our arm system) which is now in our new flash.bin image, meaning now we have our own embedded application which can be placed and boot using the U-Boot. To run the test application we first have to find its address by calculating the address as follows:

Run the following command on another ubuntu shell in the same directory as our flash.bin binary image:

```
#printf "bootm 0x%X\n" $(expr $(stat -c%s u-boot.bin) + 65536)
```

The above command takes the size of u-boot.bin (argument: \$(expr \$(stat -c%s u-boot.bin)) and sum the initial address where flash.bin (65536) is mapped.

What's the outcome address? (2 marks)

Take the address and confirm that it is actually the test.bin image you created by executing the following command on our flash.bin image prompt:

```
versatilepb$ iminfo [address]
```

Is that our test.bin image memory address? How do you know? (3 marks)

After confirming the image, boot the application using the bootm command and the address as the argument. What's the outcome? Show Screenshot. (5 marks)

Implement two functions **void print_uart1(const char s)** and **static inline char upperchar(char c)** that prints a single character and that turns a given character to its upper case respectively. Use the two printing out functions to display a character and its uppercase on the console. That is cross compile your ARM program as explained in the test program above and boot using the "bootm" command of the u-boot. (5)

HINT: To get an uppercase of any given lowercase letter, we first check whether that letter falls within the range of alphabetical letter (i.e give-letter>='a' and given-letter<='z'), then we subtract that letter from the sum of character 'a' and its uppercase, otherwise print the letter as it is.

Press "**Ctrl + a**" and then press "**x**" to exit QEMU.

4. SQLite as an Embedded DBMS (20)

Consider you are developing an embedded system for a timber company that produces high quality wood plank. These planks are then used to construct exclusive furniture, sculptures and other wooden products. The embedded system you are developing is responsible for monitoring and controlling the environmental conditions in which the wood is curing. The wooden planks cures in huge warehouses (think big garages). The plans are placed on shelves, which are formed into vertical collections of racks (ie. one rack is a stack of shelves in this case). Each rack has one or more humidity and temperature (HT) sensors (usually two per rack). Your embedded controller receives humidity and temperature readings every couple of minutes for each sensor. The controller turns on heaters, fans and closes/shuts vents as needed to try to maintain a specified temperature and humidity readings for particular racks (i.e. each rack has a desired H & T value).

The following are the sample tables populated with some data for the above problem description:

Rack Table:

rackID	TreeType	Time
1	Mahagony	09:30
2	Pine	10:24
3	Spruce	10:25
4	Cetrus	11:00
5	Lyptus	07:00
7	Blue gum	16:00

Sensor Conditions table:

sensorID	Time	Humidity	Temperature
1	10:00	9	24.7
2	9:30	10	25
3	10:24	10	27
4	16:01	11	21
5	11:00	8	20

Sensor assignment table:

sensorAssignmentID	sensorID	rackID
1	3	2
2	2	4
3	4	5
4	1	1
5	5	3

SQLite is one of the open source embedded database management systems. Help on SQLite can be found here: <http://www.sqlite.org/sqlite.html>. One can start SQLite using the command line on linux (Ubuntu In the Blue Lab), ***sqlite [dbname]***, with **[dbname]** replaced by the name of the database. In this part of the prac, you are required to:

1. Design a database Entity Relational diagram for the schema described above for the previously described problem using the basic types of relationships, called the cardinality of the relationship (1:1 - The one to one relationship, 1:M - The one to many relationship, M:M - The many to many relationship). (10)
2. Using the help from the SQLite website help above, create a database named **sensordb** and create the tables **Rack**, **SensorCondition**, and **SensorAssignment** and populate the table with the sample data in the above tables using an INSERT keyword (Check the sample.sql script given as sample) of the SQLite database. (5)
3. Using SQLite SELECT keyword, what type of wood(tree type) and conditions (Temperature and humidity) were on rack 2 at 10:00? (5)

END OF PRACTICAL
