



Kernel - Cross-compiling

Objective: Learn how to cross-compile a kernel for another target platform.

After this lab, you will be able to

- Set up a cross-compiling environment
- Configure the kernel `Makefile` accordingly
- Cross compile the kernel for the Calao arm platform
- Use U-Boot to download the kernel
- Check that the kernel you compiled can boot the system

Setup

Go to the `/home/<user>/felabs/sysdev/xkernel` directory.

If you haven't done the previous labs, install the following packages: `libqt3-mt-dev`, `g++`

Also install `uboot-mkimage`.

Target system

We are going to cross-compile and boot a Linux kernel for the ARM Calao USB-A9263 board.

Getting the sources

We are going to use the Linux 2.6.37 sources for this lab. This time, we will use `ketchup` to fetch these sources.

First, create an empty `linux-2.6.37` directory and go into it. Now, run:

```
ketchup -G 2.6.37
```

Cross-compiling environment setup

To cross-compile Linux, you need to install the cross-compiling toolchain. If a previous lab allowed you to produce the cross-compiling toolchain, we just need to make it available in the `PATH`.

Let's set the `PATH` in a permanent way by adding the below line to your `~/.bashrc` file: (all in one line)

```
export PATH=/usr/local/xtools/arm-unknown-linux-uclibcgnueabi/bin:$PATH
```

Now, every time you open a new shell, you will have the correct `PATH` setting.

If you didn't build your own toolchain, download the toolchain from <http://free-electrons.com/labs/tools/arm-unknown-linux-uclibc-gcc-4.3.2.tar.bz2> and uncompress it as root in `/`. Then, add the cross-compiler directory to the `PATH` as shown above.

Makefile setup

Modify the toplevel `Makefile` file to cross-compile for the arm platform using the above toolchain.

`Ketchup` would refuse to run in a non-empty directory which doesn't contain Linux sources.



Linux kernel configuration

By running `make help`, find the proper Makefile target to configure the kernel for the Calao USB A9263 board. Once found, use this target to configure the kernel with the ready-made configuration.

Don't hesitate to visualize the new settings by running `make xconfig` afterwards!

Cross compiling

You're now ready to cross-compile your kernel. Simply run:

```
make
```

and wait a while for the kernel to compile.

Look at the end of the kernel build output to see which file contains the kernel image.

However, the default image produced by the kernel build process is not suitable to be booted from U-Boot. A post-processing operation must be performed using the `mkimage` tool provided by U-Boot developers. This tool has already been installed in your system as part of the `uboot-mkimage` package. To run the post-processing operation on the kernel image, simply run:

```
make uImage.
```

Integrating the root filesystem

You need a root filesystem to boot from. Using the `initramfs` technique, use the root filesystem in `data/rootfs/`, and recompile your kernel.

Booting the kernel

To boot the kernel, it must first be downloaded through TFTP. Copy the `arch/arm/boot/uImage` file resulting from the kernel compilation to home directory of the TFTP server. Then, from U-Boot, download this file to the board:

```
tftp 0x21000000 uImage
```

You can now boot the kernel by telling U-Boot to start the application image at `0x21000000`:

```
bootm 0x21000000
```

U-Boot will automatically load the kernel at the right location, for which it was compiled (`0x20008000`, as specified in the U-Boot header of the `uImage` file).

You should see your kernel boot, and finally a shell prompt, which means that userspace applications are properly running on your board!

Checking the kernel version

It is sometimes useful to check that the kernel that you are running is really the one that you have just compiled. It's because errors happen. You could be booting from flash by mistake, or you could have forgotten to copy your new kernel image to the tftp server root directory.

See how simple this is compared to having to create your own filesystem!



Display the contents of the `/proc/version` file: you will see that it shows the kernel compile date. Check that this corresponds to the date of the `arch/arm/boot/uImage` file in your kernel source directory.

Flashing the kernel

In order to let the kernel boot on the board autonomously, we must flash it in the NAND flash available on the Calao platform. The NAND flash can be manipulated in U-Boot using the `nand` command, which features several subcommands. Run `help nand` to see the available subcommands and their options.

The NAND flash is logically split in three partitions by the Linux kernel, as defined in the `ek_nand_partition` definition in the `arch/arm/mach-at91/board-usb-a9263.c` board-specific file. The first partition, of 16 megabytes is reserved for the kernel.

So, let's start by erasing the first 16 megabytes of the NAND:

```
nand erase 0 0x1000000
          (start) (length)
```

Then, download the kernel into memory:

```
tftp 0x21000000 uImage
```

At the end of the download, it will print the size of the downloaded kernel image, which will be useful to know how many bytes must be flashed:

```
Bytes transferred = 3301444 (326044 hex)
```

Then, flash the kernel image by rounding the kernel size at least up to the next page (in terms of 2048 bytes pages):

```
nand write 0x21000000 0 0x327000
          (target-RAM-address) (source-nand-address) (length)
```

Then, we can boot the kernel from the NAND using:

```
nboot 0x21000000 0
      (target-ram-address) (source-nand-address)
```

This command copies data found at address 0 in nand flash, to address 0x21000000 in RAM. It finds the number of bytes to copy from nand to RAM in the `uImage` header at the beginning of the kernel image.

```
bootm 0x21000000
```

To automate this process at boot time, we need to set two environment variables:

- `bootcmd`, which contains the command run at boot time by the bootloader;
- `autostart`, which tells whether the `nboot` command should start the kernel after loading or not.

To set these variables, do:

```
setenv bootcmd nboot 0x21000000 0
setenv autostart yes
saveenv
```

This is one of the reasons why a `uImage` container is needed for Linux kernel images handled by U-boot.



Then, restart the board, and wait until the expiration of the timeout. The default boot command stored in `bootcmd` will be executed, and your kernel will start!

Going further

We want to measure how fast the bootloader and kernel execute. Download the `grabserial` program (<http://elinux.org/Grabserial>), and use it instead of `Minicom` to capture the messages coming from the serial line and add timestamps to them.

First, install the Python module required by `grabserial`:

```
sudo apt-get install python-serial
```

Exit from `Minicom` if it was still running, and run `grabserial`:

```
grabserial -t -d "/dev/ttyUSB1" -e 30 > boot.log
```

Looking at `boot.log`, find out how much time it took to reach the shell command line.

Now, recompile your kernel with LZO kernel and `initramfs` compression (find the corresponding options in the configuration interface). LZO's compression rate is slightly worse than `gzip`'s (the default compressor), but it decompresses much faster.

Boot your new kernel, and see if LZO compression reduced boot time.