Embedded Linux Training

Lab Book

Endocode AG
https://endocode.com

May 9, 2016

About this document

Updates to this document can be found on https://github.com/endocode/embedded-linux-labs.

This document was generated from LaTeX sources found on http://git.free-electrons.com/training-materials.

Copying this document

Training setup

Download files and directories used in practical labs

Install lab data

For the different labs in this course, your instructor has prepared a set of data (kernel images, kernel configurations, root filesystems and more). Download and extract its tarball from a terminal:

```
cd
wget https://github.com/endocode/embedded-linux-labs/raw/master/embedded-linux-labs.tar.xz
tar -xJvf embedded-linux-labs.tar.xz
```

Lab data are now available in an `embedded-linux-training-labs` directory in your home directory. For each lab there is a directory containing various data. This directory will also be used as working space for each lab, so that the files that you produce during each lab are kept separate.

You are now ready to start the real practical labs!

Install extra packages

Ubuntu comes with a very limited version of the `vi` editor. Install `vim`, a improved version of this editor.

```
sudo apt-get install vim
```

More guidelines

Can be useful throughout any of the labs

- Read instructions and tips carefully. Lots of people make mistakes or waste time because they missed an explanation or a guideline.
- Always read error messages carefully, in particular the first one which is issued. Some people stumble on very simple errors just because they specified a wrong file path and didn't pay enough attention to the corresponding error message.
- Never stay stuck with a strange problem more than 5 minutes. Show your problem to your colleagues or to the instructor.
- You should only use the `root` user for operations that require super-user privileges, such as: mounting a file system, loading a kernel module, changing file ownership, configuring the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.
- If you ran commands from a root shell by mistake, your regular user may no longer be able to handle the corresponding generated files. In this case, use the `chown -R` command

to give the new files back to your regular user.
Example: `chown -R myuser.myuser linux-3.4`

Building a cross-compiling toolchain

Objective: Learn how to compile your own cross-compiling toolchain for the uClibc C library

After this lab, you will be able to:

- Configure the *crosstool-ng* tool
- Execute *crosstool-ng* and build up your own cross-compiling toolchain

Setup

Go to the `$HOME/embedded-linux-labs/toolchain` directory.

Install needed packages

Install the packages needed for this lab:

```
sudo apt-get install autoconf automake libtool libexpat1-dev \
     libncurses5-dev bison flex patch curl cvs texinfo git bc \
     build-essential subversion gawk python-dev gperf unzip \
     pkg-config wget help2man
```

Tip: instead of typing all the above package names, fetch the electronic version of these instructions and copy/paste the long command line!

Getting Crosstool-ng

Let's download Crosstool-ng through its git source repository:

```
git clone git://crosstool-ng.org/crosstool-ng
cd crosstool-ng/
git checkout -b work c4a1428ab
```

Note that if *cloning* through `git://` doesn't work due to network restrictions, you can clone through `http://` instead.

Installing Crosstool-ng

We can either install Crosstool-ng globally on the system, or keep it locally in its download directory. We'll choose the latter solution. As documented in `docs/2\ -\ Installing\ crosstool-NG.txt`, do:

```
autoreconf
./configure --enable-local
make
make install
```

Then you can get Crosstool-ng help by running


```
./ct-ng help
```

Configure the toolchain to produce

A single installation of Crosstool-ng allows to produce as many toolchains as you want, for different architectures, with different C libraries and different versions of the various components.

Crosstool-ng comes with a set of ready-made configuration files for various typical setups: Crosstool-ng calls them *samples*. They can be listed by using `./ct-ng list-samples`.

We will use the `arm-cortexa5-linux-uclibcgnueabihf` sample. It can be loaded by issuing:

```
./ct-ng arm-cortexa5-linux-uclibcgnueabihf
```

Then, to refine the configuration, let's run the `menuconfig` interface:

```
./ct-ng menuconfig
```

In `Path and misc options`:

- Change `Prefix directory` to `/usr/local/xtools/${CT_TARGET}`. This is the place where the toolchain will be installed.
- Change `Maximum log level to see` to `DEBUG` so that we can have more details on what happened during the build in case something went wrong.

In `Toolchain options`:

- Set `Tuple's alias` to `arm-linux`. This way, we will be able to use the compiler as `arm-linux-gcc` instead of `arm-cortexa5-linux-uclibcgnueabihf-gcc`, which is much longer to type.

In `C-library`:

- Enable the `IPv6` support.

In `Debug facilities`:

- Make sure that `gdb`, `strace` and `ltrace`) are enabled.
- Remove the remaining options (`dmalloc` and `duma`).
- In `gdb` options, make sure that the `Cross-gdb` and `Build a static gdbserver` options are enabled and all the other options are disabled.

Explore the different other available options by traveling through the menus and looking at the help for some of the options. Don't hesitate to ask your trainer for details on the available options. However, remember that we tested the labs with the configuration described above. You might waste time with unexpected issues if you customize the toolchain configuration.

Produce the toolchain

First, create the directory `/usr/local/xtools/` and change its owner to your user, so that Crosstool-ng can write to it.

Then, create the directory `$HOME/src` in which Crosstool-NG will save the tarballs it will download.

Nothing is simpler:

```
./ct-ng build
```


And wait!

Known issues

Source archives not found on the Internet

It is frequent that Crosstool-ng aborts because it can't find a source archive on the Internet, when such an archive has moved or has been replaced by more recent versions. New Crosstool-ng versions ship with updated URLs, but in the meantime, you need work-arounds.

If this happens to you, what you can do is look for the source archive by yourself on the Internet, and copy such an archive to the `src` directory in your home directory. Note that even source archives compressed in a different way (for example, ending with `.gz` instead of `.bz2`) will be fine too. Then, all you have to do is run `./ct-ng build` again, and it will use the source archive that you downloaded.

ppl-0.10.2 compiling error with gcc 4.7.1

If you are using gcc 4.7.1, for example in Ubuntu 12.10 (not officially supported in these labs), compilation will fail in `ppl-0.10.2` with the below error:

```
error: 'f_info' was not declared in this scope
```

One solution is to add the `-fpermissive` flag to the `CT_EXTRA_FLAGS_FOR_HOST` setting (in `Path and misc options -> Extra host compiler flags`).

Testing the toolchain

You can now test your toolchain by adding `/usr/local/xtools/arm-cortexa5-linux-uclibcgnueabihf/bin/` to your `PATH` environment variable and compiling the simple `hello.c` program in your main lab directory with `arm-linux-gcc`.

You can use the `file` command on your binary to make sure it has correctly been compiled for the ARM architecture.

Cleaning up

To save about 6 GB of storage space, do a `./ct-ng clean` in the Crosstool-NG source directory. This will remove the source code of the different toolchain components, as well as all the generated files that are now useless since the toolchain has been installed in `/usr/local/xtools`.

Bootloader - U-Boot

Objectives: Set up serial communication, compile and install the U-Boot bootloader, use basic U-Boot commands, set up TFTP communication with the development workstation.

As the bootloader is the first piece of software executed by a hardware platform, the installation procedure of the bootloader is very specific to the hardware platform. There are usually two cases:

- The processor offers nothing to ease the installation of the bootloader, in which case the JTAG has to be used to initialize flash storage and write the bootloader code to flash. Detailed knowledge of the hardware is of course required to perform these operations.
- The processor offers a monitor, implemented in ROM, and through which access to the memories is made easier.

The Xplained board, which uses the SAMA5D3 SoCs, falls into the second category. The monitor integrated in the ROM reads the MMC/SD card to search for a valid bootloader before looking at the internal NAND flash for a bootloader. In case nothing is available, it will operate in a fallback mode, that will allow to use an external tool to reflash some bootloader through USB. Therefore, either by using an MMC/SD card or that fallback mode, we can start up a SAMA5D3-based board without having anything installed on it.

Downloading Atmel's flashing tool

Go to the `~/embedded-linux-labs/bootloader` directory.

We're going to use that fallback mode, and its associated tool, `sam-ba`.

We first need to download this tool, from Atmel's website[1].

```
wget http://www.atmel.com/Images/sam-ba_2.15.zip
unzip sam-ba_2.15.zip
```

To run `sam-ba`, you will need to install the below libraries:

```
sudo apt-get install libxss1 libxft2
```

Setting up serial communication with the board

Plug the USB-to-serial cable on the Xplained board. The blue end of the cable is going to `GND` on `J23`, red on `RXD` and green on `TXD`. When plugged in your computer, a serial port should appear, `/dev/ttyUSB0`.

You can also see this device appear by looking at the output of `dmesg`.

To communicate with the board through the serial port, install a serial communication program, such as `picocom`:

[1] In case this website is down, you can also find this tool on http://free-electrons.com/labs/tools/.


```
sudo apt-get install picocom
```

You also need to make your user belong to the `dialout` group to be allowed to write to the serial console:

```
sudo adduser $USER dialout
```

You need to log out and in again for the group change to be effective.

Run `picocom -b 115200 /dev/ttyUSB0`, to start serial communication on `/dev/ttyUSB0`, with a baudrate of 115200.

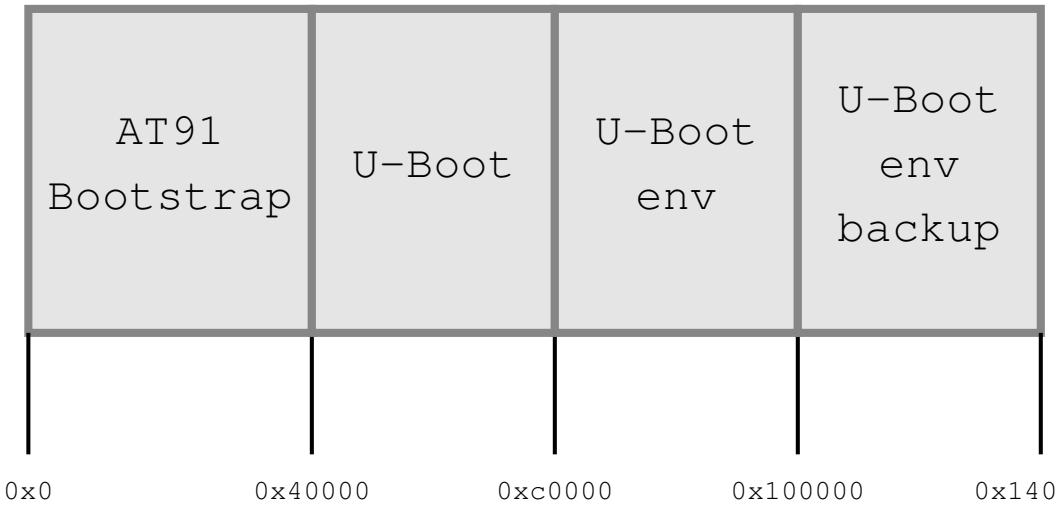
You can now power-up the board by connecting the micro-USB cable to the board, and to your PC at the other end. If a system was previously installed on the board, you should be able to interact with it through the serial line.

If you wish to exit `picocom`, press `[Ctrl][a]` followed by `[Ctrl][x]`.

AT91Bootstrap Setup

The boot process is done in two steps with the ROM monitor trying to execute a first piece of software, called `AT91Bootstrap`, from its internal SRAM, that will initialize the DRAM, load `U-Boot` that will in turn load Linux and execute it.

As far as bootloaders are concerned, the layout of the NAND flash will look like:



- Offset `0x0` for the first stage bootloader is dictated by the hardware: the ROM code of the SAMA5D3 looks for a bootloader at offset `0x0` in the NAND flash.
- Offset `0x40000` for the second stage bootloader is decided by the first stage bootloader. This can be changed by changing the AT91Bootstrap configuration.
- Offset `0xc0000` of the U-Boot environment is decided by U-Boot. This can be changed by modifying the U-Boot configuration.

The first item to compile is AT91Bootstrap that you can fetch from Atmel's GitHub account:

```
git clone git://github.com/linux4sam/at91bootstrap.git
cd at91bootstrap
git checkout v3.7.1
```


Then, we first need to configure the build system for our setup. We're going to need a few pieces of information for this:

- Which board you want to run AT91Bootstrap on
- Which device should AT91Bootstrap will be stored on
- What component you want AT91Boostrap to load

You can get the list of the supported boards by listing the `board` directory. You'll see that in each of these folders, we have a bunch of `defconfig` files, that are the supported combinations. In our case, we will load U-Boot, from NAND flash (`nf` in the `defconfig` file names).

After finding the right `defconfig` file, load it using `make <defconfig_filename>` (just the file name, without the directory part).

In recent versions of AT91Bootstrap, you can now run `make menuconfig` to explore options available in this program.

The next thing to do is to specific the cross-compiler prefix (the part before `gcc` in the cross-compiler executable name):

```
export CROSS_COMPILE=arm-linux-
```

You can now start compiling using `make`[2].

At the end of the compilation, you should have a file called `sama5d3_xplained-nandflashboot-uboot-*.bin`, in the `binaries` folder.

In order to flash it, we need to do a few things. First, remove the `NAND CS` jumper on the board. It's next to the pin header closest to the Micro-USB plug. Now, press the `RESET` button. On the serial port, you should see `RomBoot`.

Put the jumper back.

Then, start `sam-ba` (or `sam-ba_64` if using a 64 bit installation of Ubuntu). Run the executable from where it was extracted. You'll get a small window. Select the `ttyACM0` connection, and the `at91sama5d3x-ek` board. Hit `Connect`.

You need to:

- Hit the `NANDFlash` tab
- In the `Scripts` choices, select `Enable NandFlash` and hit `Execute`
- Select `Erase All`, and execute the command
- Then, select and execute `Enable OS PMECC parameters` in order to change the NAND ECC parameters to what RomBOOT expects. Change the number of ECC bits to `4`, and the ECC offset to `36`.
- Finally, send the image we just compiled using the command `Send Boot File`

AT91Bootstrap should be flashed now, keep `sam-ba` open, and move to the next section.

U-Boot setup

Download U-Boot:

```
wget ftp://ftp.denx.de/pub/u-boot/u-boot-2015.04.tar.bz2
```

[2] You can speed up the compiling by using the `-jX` option with `make`, where X is the number of parallel jobs used for compiling. Twice the number of CPU cores is a good value.

We're going to use a specific U-Boot version, `2015.04`, which we have tested to work on the Atmel Xplained board. More recent versions may also work, but we have not tested them.

Extract the source archive and get an understanding of U-Boot's configuration and compilation steps by reading the `README` file, and specifically the *Building the Software* section.

Basically, you need to:

- Set the `CROSS_COMPILE` environment variable;
- Run `make <NAME>_defconfig`, where `<NAME>` is the name of your board as declared in the directory `configs/`. There are two flavors of the Xplained configuration: one to run from the SD card (`sama5d3_xplained_mmc`) and one to run from the NAND flash (`sama5d3_xplained_nandflash`). Since we're going to boot on the NAND, use the latter. Note that for our platform, both these choices are sharing most of their configuration, that is defined in `include/configs/sama5d3_xplained.h`. Read this file to get an idea of how a U-Boot configuration file is written;
- Now that you have a valid initial configuration, you can now run `make menuconfig` to further edit your bootloader features.
- Finally, run `make`, which should build U-Boot.

Now, in `sam-ba`, in the `Send File Name` field, set the path to the `u-boot.bin` that was just compiled, and set the address to `0x40000`. Click on the `Send File` button.

You can now exit `sam-ba`.

Testing U-Boot

Reset the board and check that it boots your new bootloaders. You can verify this by checking the build dates:

```
AT91Bootstrap 3.7.1 (Wed Oct 28 06:48:23 CET 2015)

NAND: ONFI flash detected
NAND: Manufacturer ID: 0x2c Chip ID: 0x32
NAND: Disable On-Die ECC
NAND: Initialize PMECC params, cap: 0x4, sector: 0x200
NAND: Image: Copy 0x80000 bytes from 0x40000 to 0x26f00000
NAND: Done to load image


U-Boot 2015.04 (Oct 28 2015 - 07:11:52)

CPU: SAMA5D36
Crystal frequency:       12 MHz
CPU clock        :      528 MHz
Master clock     :      132 MHz
DRAM:  256 MiB
NAND:  256 MiB
MMC:   mci: 0
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
```

```
Err:   serial
Net:   gmac0
Error: gmac0 address not set.
, macb0
Error: macb0 address not set.

Hit any key to stop autoboot:  0
```

Interrupt the countdown to enter the U-Boot shell:

```
U-Boot #
```

In U-Boot, type the `help` command, and explore the few commands available.

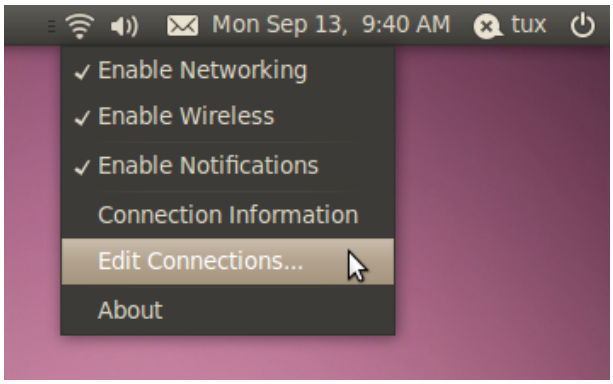
Setting up Ethernet communication

Later on, we will transfer files from the development workstation to the board using the TFTP protocol, which works on top of an Ethernet connection.

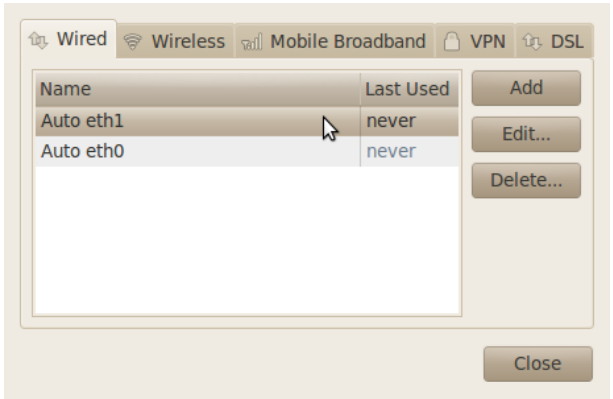
To start with, install and configure a TFTP server on your development workstation, as detailed in the bootloader slides.

With a network cable, connect the Ethernet port labelled ETH0/GETH of your board to the one of your computer.

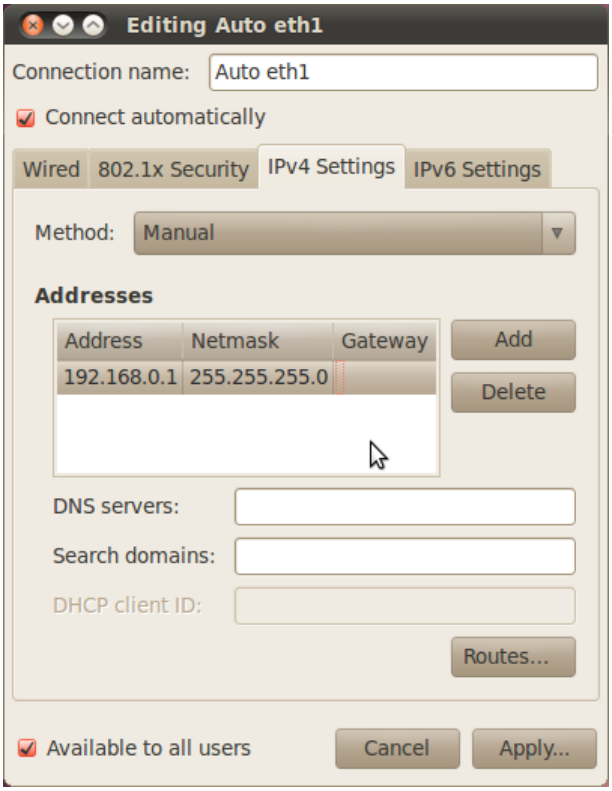
To configure this network interface on the workstation side, click on the *Network Manager* tasklet on your desktop, and select *Edit Connections*.



Select the new *wired network connection*:



In the `IPv4 Settings` tab, press the `Add` button and then choose the `Manual` method to make the interface use a static IP address, like `192.168.0.1` (of course, make sure that this address belongs to a separate network segment from the one of the main company network).



You can use `255.255.255.0` as `Netmask`, and leave the `Gateway` field untouched (if you click on the `Gateway` box, you will have to type a valid IP address, otherwise you won't be allowed to click on the `Apply` button).

Now, configure the network on the board in U-Boot by setting the `ipaddr` and `serverip` environment variables:

```
setenv ipaddr 192.168.0.100
setenv serverip 192.168.0.1
```

The first time you use your board, you also need to set the MAC address in U-boot:

```
setenv ethaddr 12:34:56:ab:cd:ef
```

In case the board was previously configured in a different way, we also turn off automatic booting after commands that can be used to copy a kernel to RAM:

```
setenv autostart no
```

To make these settings permanent, save the environment:

```
saveenv
```

Now reset your board[3].

[3] Resetting your board is needed to make your `ethaddr` permanent, for obscure reasons. If you don't, U-boot will complain that `ethaddr` is not set.

You can then test the TFTP connection. First, put a small text file in the directory exported through TFTP on your development workstation. Then, from U-Boot, do:

```
tftp 0x22000000 textfile.txt
```

> **Caution: known issue in Ubuntu 12.04 and later (still present in Ubuntu 14.04)**: if download through tftp doesn't work, you may have to stop the `tftpd-hpa` server and start it again every time you boot your workstation:
> `sudo service tftpd-hpa restart`
> The problem is Ubuntu starts this server too early, before the preconditions it needs are met. When you restart the service long after the machine has booted, this issue is no longer present. If it still doesn't work, another (radical!) workaround is to reinstall the `tftpd-hpa` server!
> `sudo apt-get install --reinstall tftpd-hpa`
> So far, we haven't had the time yet to investigate the root cause of the issue that is addressed by this last workaround.

The `tftp` command should have downloaded the `textfile.txt` file from your development workstation into the board's memory at location `0x22000000`[4].

You can verify that the download was successful by dumping the contents of the memory:

```
md 0x22000000
```

We will see in the next labs how to use U-Boot to download, flash and boot a kernel.

Rescue binaries

If you have trouble generating binaries that work properly, or later make a mistake that causes you to loose your bootloader binaries, you will find working versions under `data/` in the current lab directory.

[4] This location is part of the board DRAM. If you want to check where this value comes from, you can check the Atmel SAMA5D3 datasheet at http://www.atmel.com/tools/ATSAMA5D3-XPLD.aspx, following the *Documents* link. It's a big document (more than 1,800 pages). In this document, look for `Memory Mapping` and you will find the SoC memory map. You will see that the address range for the memory controller (*DDRC S*) starts at `0x20000000` and ends at `0x3fffffff`. This shows that the `0x22000000` address is within the address range for RAM. You can also try with other values in the same address range, knowing that our board only has 256 MB of RAM (that's `0x10000000`, so the physical RAM probably ends at `0x30000000`).

Kernel sources

Objective: Learn how to get the kernel sources and patch them.

After this lab, you will be able to:

- Get the kernel sources from the official location
- Apply kernel patches

Setup

Create the `$HOME/embedded-linux-labs/kernel` directory and go into it.

Get the sources

Go to the Linux kernel web site (`http://www.kernel.org/`) and identify the latest stable version.

Just to make sure you know how to do it, check the version of the Linux kernel running on your machine.

We will use `linux-4.4.x`, which this lab was tested with.

To practice with the `patch` command later, download the full 4.3 sources. Unpack the archive, which creates a `linux-4.3` directory. Remember that you can use `wget <URL>` on the command line to download files.

Apply patches

Download the 2 patch files corresponding to the latest 4.4 stable release: a first patch to move from 4.3 to 4.4 and a second patch to move from 4.4 to 4.4.x.

Without uncompressing them (!), apply the 2 patches to the Linux source directory.

View one of the 2 patch files with `vi` or `gvim` (if you prefer a graphical editor), to understand the information carried by such a file. How are described added or removed files?

Rename the `linux-4.3` directory to `linux-4.4.<x>`.

Kernel - Cross-compiling

Objective: Learn how to cross-compile a kernel for an ARM 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 Atmel SAMA5D3 Xplained ARM board
- Use U-Boot to download the kernel
- Check that the kernel you compiled starts the system

Setup

Go to the `$HOME/embedded-linux-labs/kernel` directory.

Install the package `libqt4-dev` which is needed for the `xconfig` kernel configuration interface.

Target system

We are going to cross-compile and boot a Linux kernel for the Atmel SAMA5D3 Xplained board.

Kernel sources

We will re-use the kernel sources downloaded and patched in the previous lab.

Cross-compiling environment setup

To cross-compile Linux, you need to have a cross-compiling toolchain. We will use the cross-compiling toolchain that we previously produced, so we just need to make it available in the PATH:

```
export PATH=/usr/local/xtools/arm-cortexa5-linux-uclibcgnueabihf/bin:$PATH
```

Also, don't forget to either:

- Define the value of the `ARCH` and `CROSS_COMPILE` variables in your environment (using `export`)
- **Or** specify them on the command line at every invocation of `make`, i.e: `make ARCH=... CROSS_COMPILE=... <target>`

Linux kernel configuration

By running `make help`, find the proper Makefile target to configure the kernel for the Xplained board (hint: the default configuration is not named after the board, but after the SoC name). 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!

In the kernel configuration, as an experiment, change the kernel compression from Gzip to XZ. This compression algorithm is far more efficient than Gzip, in terms of compression ratio, at the expense of a higher decompression time.

Cross compiling

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

```
make
```

and wait a while for the kernel to compile. Don't forget to use `make -j<n>` if you have multiple cores on your machine!

Look at the end of the kernel build output to see which file contains the kernel image. You can also see the Device Tree `.dtb` files which got compiled. Find which `.dtb` file corresponds to your board.

Copy the linux kernel image and DTB files to the TFTP server home directory.

Load and boot the kernel using U-Boot

We will use TFTP to load the kernel image on the Xplained board:

- On your workstation, copy the `zImage` and DTB files to the directory exposed by the TFTP server.
- On the target (in the U-Boot prompt), load `zImage` from TFTP into RAM at address 0x21000000:
  `tftp 0x21000000 zImage`
- Now, also load the DTB file into RAM at address 0x22000000:
  `tftp 0x22000000 at91-sama5d3_xplained.dtb`
- Boot the kernel with its device tree:
  `bootz 0x21000000 - 0x22000000`

You should see Linux boot and finally crashing. This is expected: we haven't provided a working root filesystem for our device yet.

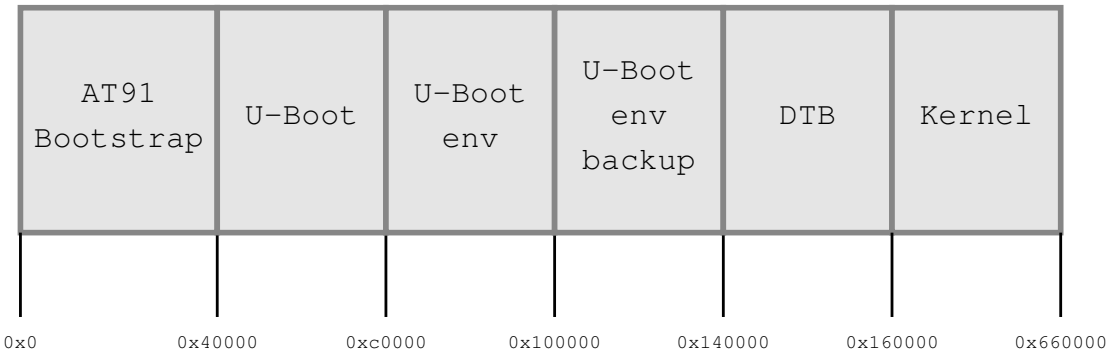
You can now automate all this every time the board is booted or reset. Reset the board, and specify a different `bootcmd`:

```
setenv bootcmd 'tftp 0x21000000 zImage; tftp 0x22000000 at91-sama5d3_xplained.dtb; bootz 0x21000000 - 0x22000000'
saveenv
```

Flashing the kernel and DTB in NAND flash

In order to let the kernel boot on the board autonomously, we can flash the kernel image and DTB in the NAND flash available on the Xplained board.

After storing the first stage bootloader, U-boot and its environment variables, we will keep special areas in NAND flash for the DTB and Linux kernel images:



So, let's start by erasing the corresponding 128 KiB of NAND flash for the DTB:

```
nand erase 0x140000 0x20000
        (NAND offset) (size)
```

Then, let's erase the 5 MiB of NAND flash for the kernel image:

```
nand erase 0x160000 0x500000
```

Then, copy the DTB and kernel binaries from TFTP into memory, using the same addresses as before.

Then, flash the DTB and kernel binaries:

```
nand write 0x22000000 0x140000 0x20000
           (RAM addr) (NAND offset) (size)
nand write 0x21000000 0x160000 0x500000
```

Power your board off and on, to clear RAM contents. We should now be able to load the DTB and kernel image from NAND and boot with:

```
nand read 0x22000000 0x140000 0x20000
          (RAM addr) (offset) (size)
nand read 0x21000000 0x160000 0x500000
bootz 0x21000000 - 0x22000000
```

Write a U-Boot script that automates the DTB + kernel download and flashing procedure. Finally, using `editenv bootcmd`, adjust `bootcmd` so that the Xplained board boots using the kernel in flash.

Now, reset the board to check that it boots fine from NAND flash. Check that this is really your own version of the kernel that's running.

