Embedded Linux Training

Lab Book

Endocode AG
https://endocode.com

January 17, 2017

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

Corrections, suggestions, contributions and translations are welcome!

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/lab/embedded-linux-labs.tgz
tar -xvf embedded-linux-labs.tgz
```

Lab data are now available in an `embedded-linux-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 the cross-compiling toolchain

In this training we use a pre-compiled cross-compiling toolchain. Install the needed packages:

```
sudo apt-get install autoconf automake libtool-bin libexpat1-dev \
             libncurses5-dev bison flex patch curl git build-essential \
             unzip wget help2man gcc-arm-linux-gnueabihf
```

Install extra packages

Feel free to install other packages you may need for your development environment. In particular, we recommend to install your favorite text editor and configure it to your taste. The favorite text editors of embedded Linux developers are of course *Vim* and *Emacs*, but there are also plenty of other possibilities, such as *GEdit*, *Qt Creator*, *CodeBlocks*, *Geany*, etc.

It is worth mentioning that by default, Ubuntu comes with a very limited version of the `vi` editor. So if you would like to use `vi`, we recommend to use the more featureful version by installing the `vim` package.

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/`
- If you are using Gnome Terminal (the default terminal emulator in Ubuntu 16.04), you can use tabs to have multiple terminals in the same window. There's no more menu option to create a new tab, but you can get one by pressing the `[Ctrl] [Shift] [t]` keys.

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`.

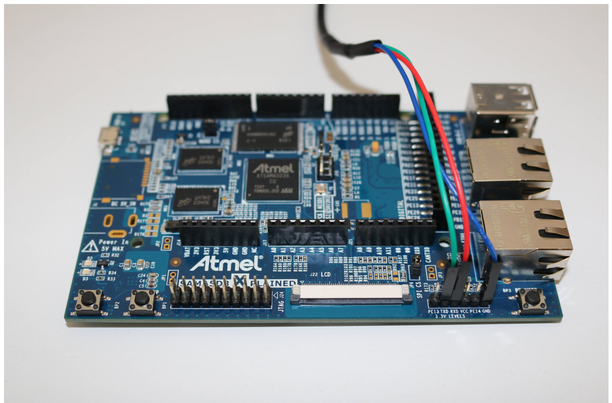
We first need to download this tool, from Atmel's website[1].

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

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`.

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



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`:

```
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.

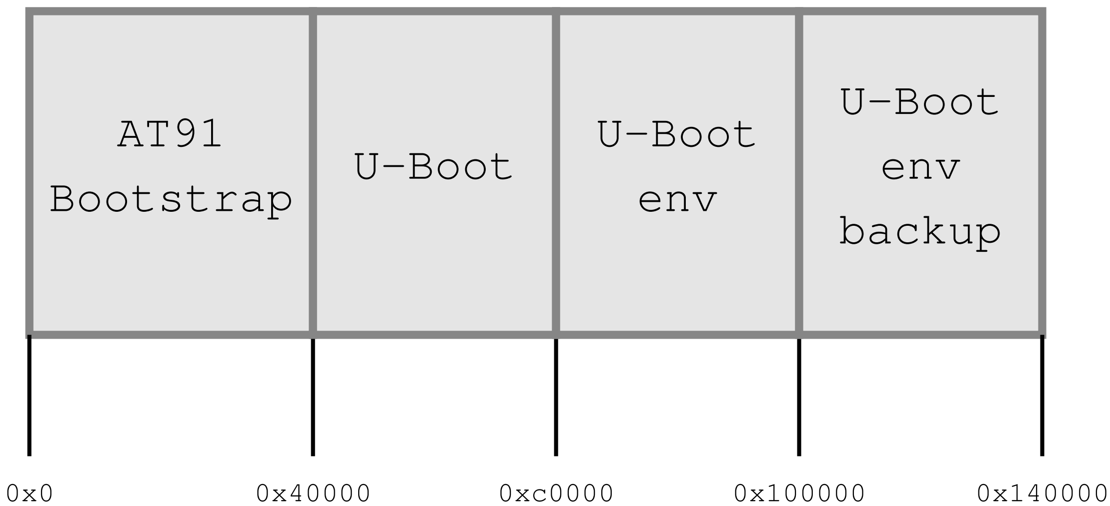
You can now power-up the board by connecting the micro-USB cable to the board, and to your PC at the other end. If a system was previously installed on the board, you should be able to interact with it through the serial line.

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 https://github.com/linux4sam/at91bootstrap.git
cd at91bootstrap
git checkout v3.8.5
```

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, using the Atmel SAMA5D3 Xplained board, we will load U-Boot, from NAND flash on (`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-gnueabihf-
```

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-xplained` 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. Select and execute `Pmecc configuration` and 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-2016.05.tar.bz2`

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 the list of available configurations can be found in the `configs/` directory. 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.
- 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`.

[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.

Testing U-Boot

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

```
AT91Bootstrap 3.8.5 (Tue May 17 12:06:03 EDT 2016)

NAND: ONFI flash detected
NAND: Manufacturer ID: 0x2c Chip ID: 0x32
NAND: Page Bytes: 0x800, Spare Bytes: 0x40
NAND: ECC Correctability Bits: 0x4, ECC Sector Bytes: 0x200
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 2016.05 (May 17 2016 - 12:41:15 -0400)

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:

```
=>
```

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.

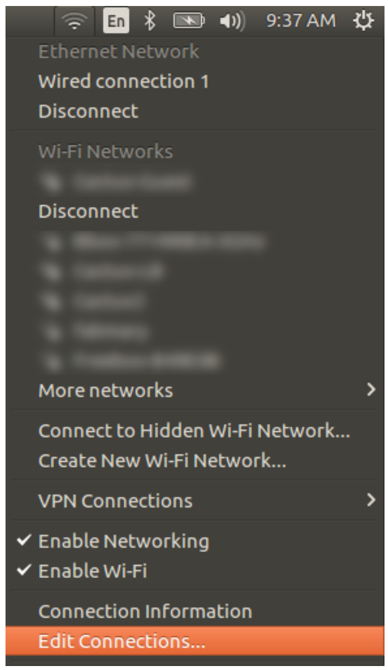
To start with, install and configure a TFTP server on your development workstation, as detailed in the bootloader slides.

```
# Install the tftpd server package
sudo apt-get install tftpd-hpa
```

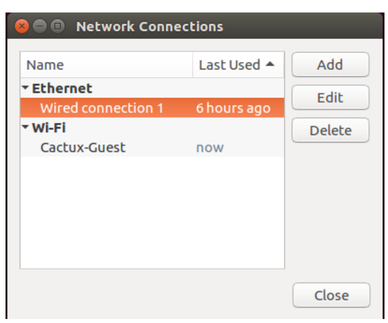
```
# Change the owner of '/var/lib/tftpboot' to allow the developer
# writing to the directory
sudo chown $USER /var/lib/tftpboot
```

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

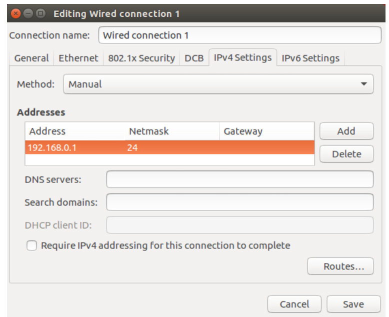
To configure this network interface on the workstation side, click on the *Network Manager* tasklet on your desktop, and select *Edit Connections*.



Select *Wired connection 1* and press the *Edit* button.



In the `IPv4 Settings` tab, 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 `24` 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 `Save` 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].

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

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

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.

[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.

[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`).

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.

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 `qt5-default` package 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.

Get the kernel 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-4.6.x`, which this lab was tested with.

Download the full 4.6 sources. Unpack the archive, which creates a `linux-4.6` directory. Remember that you can use `wget <URL>` on the command line to download files.

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 installed.

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. Hint: The filename starts with at91-sama5d....

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 panicking. This is expected: we haven't provided a working root filesystem for our device yet.

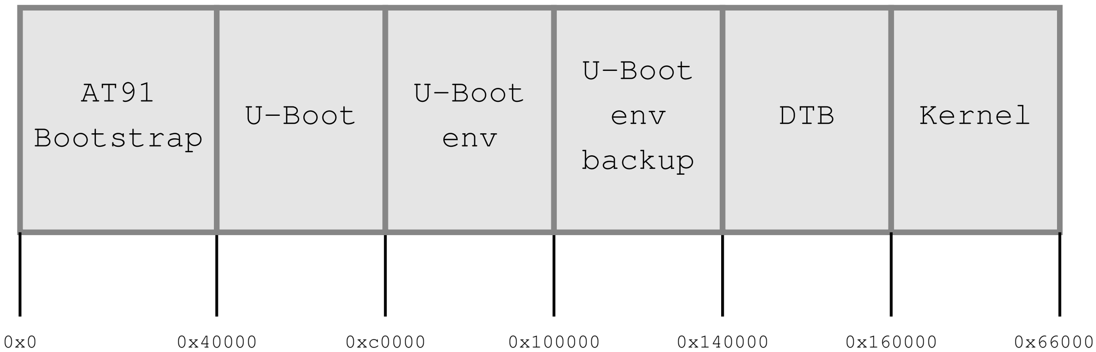
You can now automate all this every time the board is booted or reset. Reset the board, and specify a different `bootcmd`:

```
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.

You are now ready to modify `bootcmd` to boot the board from flash. But first, save the settings for booting from `tftp`:

```
setenv bootcmdtftp ${bootcmd}
```

This will be useful to switch back to `tftp` booting mode later in the labs.

Finally, using `editenv bootcmd`, adjust `bootcmd` so that the Xplained board starts using the kernel in flash.

Now, reset the board to check that it boots in the same way from NAND flash. Check that this is really your own version of the kernel that's running[5]

[5] Look at the kernel log. You will find the kernel version number as well as the date when it was compiled. That's very useful to check that you're not loading an older version of the kernel instead of the one that you've just compiled.

Tiny embedded system with BusyBox

Objective: making a tiny yet full featured embedded system

After this lab, you will:

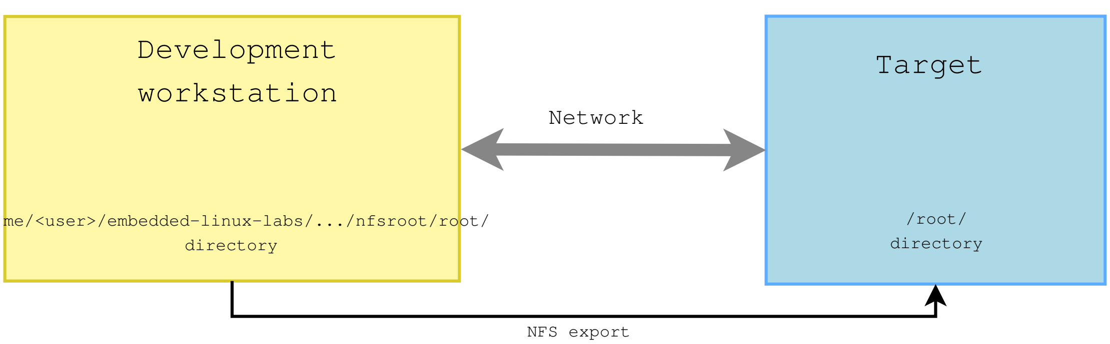
- be able to configure and build a Linux kernel that boots on a directory on your workstation, shared through the network by NFS.
- be able to create and configure a minimalistic root filesystem from scratch (ex nihilo, out of nothing, entirely hand made...) for the Xplained board
- understand how small and simple an embedded Linux system can be.
- be able to install BusyBox on this filesystem.
- be able to create a simple startup script based on /sbin/init.
- be able to set up a simple web interface for the target.
- have an idea of how much RAM a Linux kernel smaller than 1 MB needs.

Lab implementation

While (s)he develops a root filesystem for a device, a developer needs to make frequent changes to the filesystem contents, like modifying scripts or adding newly compiled programs.

It isn't practical at all to reflash the root filesystem on the target every time a change is made. Fortunately, it is possible to set up networking between the development workstation and the target. Then, workstation files can be accessed by the target through the network, using NFS.

Unless you test a boot sequence, you no longer need to reboot the target to test the impact of script or application updates.



Setup

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

Kernel configuration

We will re-use the kernel sources from our previous lab, in `$HOME/embedded-linux-labs/kernel/`.

In the kernel configuration built in the previous lab, verify that you have all options needed for booting the system using a root filesystem mounted over NFS, and if necessary, enable them and rebuild your kernel.

Setting up the NFS server

Create a `nfsroot` directory in the current lab directory. This `nfsroot` directory will be used to store the contents of our new root filesystem.

Install the NFS server by installing the `nfs-kernel-server` package if you don't have it yet. Once installed, edit the `/etc/exports` file as root to add the following line, assuming that the IP address of your board will be `192.168.0.100`:

```
/home/<user>/embedded-linux-labs/tinysystem/nfsroot 192.168.0.100(rw,no_root_squash,no_subtree_check)
```

Make sure that the path and the options are on the same line. Also make sure that there is no space between the IP address and the NFS options, otherwise default options will be used for this IP address, causing your root filesystem to be read-only.

Then, restart the NFS server:

```
sudo service nfs-kernel-server restart
```

Booting the system

First, boot the board to the U-Boot prompt. Before booting the kernel, we need to tell it that the root filesystem should be mounted over NFS, by setting some kernel parameters.

Use the following U-Boot command to do so, **in just 1 line**

```
setenv bootargs console=ttyS0,115200 root=/dev/nfs ip=192.168.0.100:::::eth0
   nfsroot=192.168.0.1:/home/<user>/embedded-linux-labs/tinysystem/nfsroot rw
```

Of course, you need to adapt the IP addresses to your exact network setup. Save the environment variables (with `saveenv`).

You will later need to make changes to the `bootargs` value. Don't forget you can do this with the `editenv` command.

Now, boot your system. The kernel should be able to mount the root filesystem over NFS:

```
VFS: Mounted root (nfs filesystem) on device 0:14.
```

If the kernel fails to mount the NFS filesystem, look carefully at the error messages in the console. If this doesn't give any clue, you can also have a look at the NFS server logs in `/var/log/syslog`.

However, at this stage, the kernel should stop because of the below issue:

```
[    7.476715] devtmpfs: error mounting -2
```

This happens because the kernel is trying to mount the *devtmpfs* filesystem in `/dev/` in the root filesystem. To address this, create a `dev` directory under `nfsroot` and reboot.

Now, the kernel should complain for the last time, saying that it can't find an init application:

```
Kernel panic - not syncing: No working init found.  Try passing init= option to kernel.
  See Linux Documentation/init.txt for guidance.
```

Obviously, our root filesystem being mostly empty, there isn't such an application yet. In the next paragraph, you will add Busybox to your root filesystem and finally make it usable.

Root filesystem with Busybox

Download the sources of the latest BusyBox 1.26.x release.

To configure BusyBox, we won't be able to use `make xconfig`, which is currently broken for BusyBox in Ubuntu (14.04 and 16.04), because of Qt library dependencies.

We are going to use `make gconfig` this time. Before doing this, install the required packages:

```
sudo apt-get install libglade2-dev
```

Now, configure BusyBox with the configuration file provided in the `data/` directory (remember that the Busybox configuration file is `.config` in the Busybox sources).

If you don't use the BusyBox configuration file that we provide, at least, make sure you build BusyBox statically! Compiling Busybox statically in the first place makes it easy to set up the system, because there are no dependencies on libraries. Later on, we will set up shared libraries and recompile Busybox.

Build BusyBox using the toolchain that you used to build the kernel.

Going back to the BusyBox configuration interface specify the installation directory for BusyBox[6]. It should be the path to your `nfsroot` directory.

Now run `make install` to install BusyBox in this directory.

Try to boot your new system on the board. You should now reach a command line prompt, allowing you to execute the commands of your choice.

Virtual filesystems

Run the `ps` command. You can see that it complains that the `/proc` directory does not exist. The ps command and other process-related commands use the `proc` virtual filesystem to get their information from the kernel.

From the Linux command line in the target, create the `proc`, `sys` and `etc` directories in your root filesystem.

Now mount the `proc` virtual filesystem. Now that `/proc` is available, test again the `ps` command.

Note that you can also now halt your target in a clean way with the `halt` command, thanks to `proc` being mounted[7].

System configuration and startup

The first user space program that gets executed by the kernel is `/sbin/init` and its configuration file is `/etc/inittab`.

In the BusyBox sources, read details about `/etc/inittab` in the `examples/inittab` file.

Then, create a `/etc/inittab` file and a `/etc/init.d/rcS` startup script declared in `/etc/inittab`. In this startup script, mount the `/proc` and `/sys` filesystems.

[6] You will find this setting in `Install Options -> BusyBox installation prefix`.

[7] `halt` can find the list of mounted filesystems in `/proc/mounts`, and unmount each of them in a clean way before shutting down.

Any issue after doing this?

Starting the shell in a proper terminal

Before the shell prompt, you probably noticed the below warning message:

```
/bin/sh: can't access tty; job control turned off
```

This happens because the shell specified in the `/etc/inittab` file in started by default in `/dev/console`:

```
::askfirst:/bin/sh
```

When nothing is specified before the leading `::`, `/dev/console` is used. However, while this device is fine for a simple shell, it is not elaborate enough to support things such as job control (`[Ctrl][c]` and `[Ctrl][z]`), allowing to interrupt and suspend jobs.

So, to get rid of the warning message, we need `init` to run `/bin/sh` in a real terminal device:

```
ttyS0::askfirst:/bin/sh
```

Reboot the system and the message will be gone!

Switching to shared libraries

Take the `hello.c` program supplied in the lab `data` directory. Cross-compile it for ARM, dynamically-linked with the libraries, and run it on the target.

You will first encounter a very misleading `not found` error, which is not because the `hello` executable is not found, but because something else is not found using the attempt to execute this executable. What's missing is the `ld-uClibc.so.0` executable, which is the dynamic linker required to execute any program compiled with shared libraries. Using the `find` command (see examples in your command memento sheet), look for this file in the toolchain install directory, and copy it to the `lib/` directory on the target.

Then, running the executable again and see that the loader executes and finds out which shared libraries are missing.

If you still get the same error message, work, just try again a few seconds later. Such a delay can be needed because the NFS client can take a little time (at most 30-60 seconds) before seeing the changes made on the NFS server.

Similarly, find the missing libraries in the toolchain and copy them to `lib/` on the target.

Once the small test program works, we are going to recompile Busybox without the static compilation option, so that Busybox takes advantages of the shared libraries that are now present on the target.

Before doing that, measure the size of the `busybox` executable.

Then, build Busybox with shared libraries, and install it again on the target filesystem. Make sure that the system still boots and see how much smaller the `busybox` executable got.

Implement a web interface for your device

Replicate `data/www/` to the `/www` directory in your target root filesystem.

Now, run the BusyBox http server from the target command line:

```
/usr/sbin/httpd -h /www/
```

It will automatically background itself.

If you use a proxy, configure your host browser so that it doesn't go through the proxy to connect to the target IP address, or simply disable proxy usage. Now, test that your web interface works well by opening `http://192.168.0.100` on the host.

See how the dynamic pages are implemented. Very simple, isn't it?

Filesystems - Flash file systems

Objective: Understand flash and flash file systems usage and their integration on the target

After this lab, you will be able to:

- Prepare filesystem images and flash them.
- Define partitions in embedded flash storage.

Setup

Stay in `$HOME/embedded-linux-labs/tinysystem`. Install the `mtd-utils` package, which will be useful to create UBIFS and UBI images.

Goals

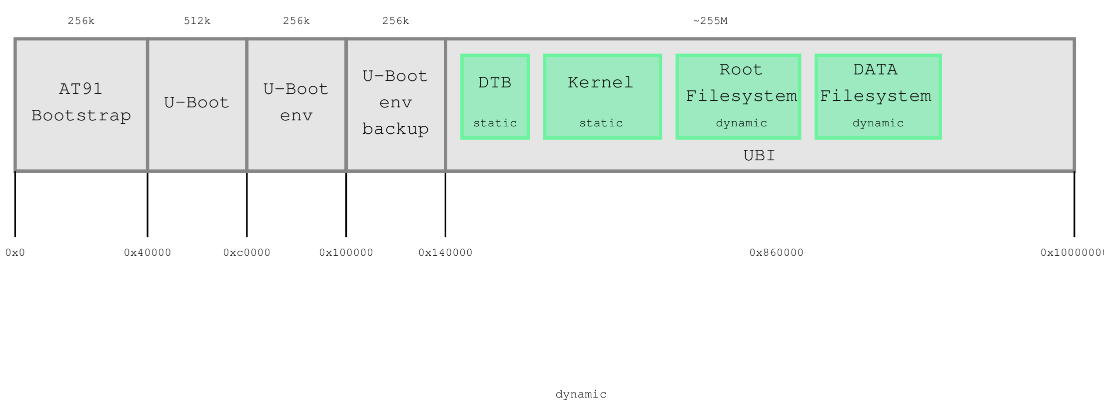
Instead of using an external MMC card as in the previous lab, we will make our system use its internal flash storage.

We will create an MTD partition to be attached to the UBI layer (the partitions previously used to store the kernel image and the DTB should be merged with this UBI partition).

The kernel and DTB images will be stored in two separate *static* (read-only) UBI volumes.

The root filesystem will be a UBI volume storing a UBIFS filesystem mounted read-only, the web server upload data will be stored in another UBI volume storing a UBIFS filesystem mounted read/write. These volumes will be *dynamic* volumes and will be 16 MiB large.

Which gives the following layout:



Enabling NAND flash and filesystems

First, make sure your kernel has support for UBI and UBIFS, and also the option allowing us to pass the partition table through the command line: (`CONFIG_MTD_CMDLINE_PARTS`).

Recompile your kernel if needed. We will update your kernel image on flash in the next section.

Filesystem image preparation

To prepare filesystem images we re-use our file-system in `$HOME/embedded-linux-labs/tinysystem/nfsroot`.

To run `mkfs.ubifs`, you will need to find the Logical Erase Block (LEB) size that UBI will use.

A solution to get such information is to list default MTD partitions on the target (`cat /proc/mtd`), and attach the last partition to UBI. In case, the last partition is `mtd5`, you will run:

```
ubiattach -m 5 /dev/ubi_ctrl
```

Doing this, you will get details in the kernel log about the MTD minimum I/O size and the LEB size that UBI will use[8].

Knowing that the `data` and `rootfs` UBI volumes will be 16 MiB big, you can now divide their total size by the LEB size, to compute the maximum of LEBs that they will contain. That's the last parameter (`-c`) that you need to pass to `mkfs.ubifs`.

You can now prepare a UBIFS filesystem image containing the files stored in the `www/upload/files` directory.

Modify the `etc/init.d/rcS` file under `nfsroot` to mount a

UBI volume called `data` [9] on `www/upload/files`.

Once done, create a UBIFS image of your root filesystem.

```
sudo mkfs.ubifs -m <min I/O unit size> -e  <LEB size> -c 1000 -r nfsroot/ -o rootfs.img
%sudo mkfs.ubifs -m 2048 -e  126976 -c 1000 -r nfsroot/ -o rootfs.img
```

UBI image preparation

Create a `ubinize` config file where you will define the 4 volumes described above, then use the `ubinize` tool to generate your UBI image. Create the config file in `$HOME/embedded-linux-labs/tinysystem/ubi.ini`.

Warning: do not use the autoresize flag (`vol_flags=autoresize`): U-Boot corrupts the UBI metadata when trying to expand the volume.

Remember that some of these volumes are static (read-only) and some are not.

Example:

```
[example-volume]
mode=ubi
image=path/to/examplefile
vol_id=1
vol_type=dynamic
vol_size=16MiB
vol_name=example
```

With the ready config file build the ubi image:

[8] Note that this command could fail if you accidently wrote to the corresponding flash blocks. If this happens, go back to U-Boot and erase NAND sectors from the starting offset of this partition: `nand erase 0x660000 0xF9A0000`

[9] We will create it when running `ubinize` in the next section


```
# sudo ubinize -o image.ubi -p <PEB size> -m <minimal I/O unit size> ubi.ini
 sudo ubinize -o image.ubi -p 128 KiB -m 2048 ubi.ini
```

MTD partitioning and flashing

Run `dmesg > /dmesg.log` to write the kernel log on the target into a file. Find the file on the development host in your nfsroot and open it in an editor. Look at the default MTD partitions in the kernel log. They do not match the way we wish to organize our flash storage. Therefore, we will define our own partitions at boot time, on the kernel command line.

Redefine the partitions in U-Boot using the `mtdids` and `mtdparts` environment variables.

Find the `devid` by running `nand info` in U-Boot. Find the `mtdid` (or Linux mtd device name) in the kernel log saved earlier.

With that informations attach the mtdid to the flash device.

```
setenv mtdids <devid>=<mtdid>
%setenv mtdids nand0=atmel_nand
```

Now define the partions for the device according to out layout.

```
setenv mtdparts mtdparts=<mtdid>:<size>(name),[<size>(name)]...
%mtdparts=atmel_nand:256k(at91bootstrap),512k(u-boot),256k(u-boot-env),256k(u-boot-env-back),-(UBI
```

Once done, execute the `mtdparts` command and check the partition definitions. The output should look like that:

```
U-Boot> mtdparts

device nand0 <atmel_nand>, # parts = 5
 #: name                size            offset          mask_flags
 0: at91bootstrap       0x00040000      0x00000000      0
 1: u-boot              0x00080000      0x00040000      0
 2: u-boot-env          0x00040000      0x000c0000      0
 3: u-boot-env-back     0x00040000      0x00100000      0
 4: UBI                 0x0fec0000      0x00140000      0

active partition: nand0,0 - (at91bootstrap) 0x00040000 @ 0x00000000

defaults:
mtdids  : none
mtdparts: none
U-Boot>
```

You can now safely erase the UBI partition without risking any corruption on other partitions.

Download the UBI image (using `tftp`) you have created in the previous section and flash it on the UBI partition.

When flashing the UBI image, use the `trimffs` version of the command `nand write`[10].

[10] The command `nand write.trimffs` skips the blank sectors instead of writing them. It is needed because the algorithm used by the hardware ECC for the SAMA5D3 SoC generates a checksum with bytes different from `0xFF` if the page is blank. Linux only checks the page, and if it is blank it doesn't erase it, but as the OOB is not blank it leads to ECC errors. More generally it is not recommended writing more than one time on a page and its OOB even if the page is blank.

Loading kernel and DTB images from UBI and booting it

From U-Boot, retrieve the kernel and DTB images from their respective UBI volumes and try to boot them. If it works, you can modify your `bootcmd` accordingly.

Set the `bootargs` variable so that:

- The `mtdparts` environment variable contents are passed to the kernel through its command line.
- The UBI partition is automatically attached to the UBI layer at boot time
- The root filesystem is mounted from the root volume, and is mounted read-only (kernel parameter `ro`).

Boot the target, and check that your system still works as expected. Your root filesystem should be mounted read-only, while the data filesystem should be mounted read-write, allowing you to upload data using the web server.

Going further

Atomic update

UBI also provides an atomic update feature, which is particularly useful if you need to safely upgrade sensitive parts of your system (kernel, DTB or rootfs).

Duplicate the kernel volume and create a U-Boot script to fallback on the second kernel volume if the first one is corrupted:

- First create a new static volume to store your kernel backup
- Flash a valid kernel on the backup volume
- Modify your `bootcmd` to fallback to the backup volume if the first one is corrupted
- Now try to update the kernel volume and interrupt the process before it has finished and see what happens (unplug the platform)
- Create a shell script to automate kernel updates (executed in Linux). Be careful, this script should also handle the case where the backup volume has been corrupted (copy the contents of the kernel volume into the backup one)

Using *squashfs* for the root filesystem

Root filesystems are often a sensitive part of your system, and you don't want it to be corrupted, hence some people decide to use a read-only file system for their rootfs and use another file system to store their auxiliary data.

`squashfs` is one of these read-only file systems. However, `squashfs` expects to be mounted on a block device.

Use the *ubiblk* layer to emulate a read-only block device on top of a static UBI volume to mount a *squashfs* filesystem as the root filesystem:

- First create a *squashfs* image with your rootfs contents
- Then create a new static volume to store your squashfs and update it with your squashfs image

- Enable and setup the *ubiblk* layer
- Boot on your new rootfs

Using a build system, example with Buildroot

Objectives: discover how a build system is used and how it works, with the example of the Buildroot build system. Build a Linux system with libraries and make it work on the board.

Setup

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

Get Buildroot and explore the source code

The official Buildroot website is available at `http://buildroot.org/`. Download the latest stable 2016.11.1 version which we have tested for this lab. Uncompress the tarball and go inside the Buildroot source directory.

Several subdirectories or files are visible, the most important ones are:

- `boot` contains the Makefiles and configuration items related to the compilation of common bootloaders (Grub, U-Boot, Barebox, etc.)
- `configs` contains a set of predefined configurations, similar to the concept of defconfig in the kernel.
- `docs` contains the documentation for Buildroot. You can start reading `buildroot.html` which is the main Buildroot documentation;
- `fs` contains the code used to generate the various root filesystem image formats
- `linux` contains the Makefile and configuration items related to the compilation of the Linux kernel
- `Makefile` is the main Makefile that we will use to use Buildroot: everything works through Makefiles in Buildroot;
- `package` is a directory that contains all the Makefiles, patches and configuration items to compile the user space applications and libraries of your embedded Linux system. Have a look at various subdirectories and see what they contain;
- `system` contains the root filesystem skeleton and the *device tables* used when a static `/dev` is used;
- `toolchain` contains the Makefiles, patches and configuration items to generate the cross-compiling toolchain.

Configure Buildroot

In our case, we would like to:

- Generate an embedded Linux system for ARM;
- Use an external toolchain instead of having Buildroot generating one for us;
- Integrate *Busybox*, *alsa-utils* and *vorbis-tools* in our embedded Linux system;
- Integrate the target filesystem into a tarball

To run the configuration utility of Buildroot, simply run:

`make menuconfig`

Set the following options. Don't hesitate to press the `Help` button whenever you need more details about a given option:

- `Target options`
  - `Target Architecture`: `ARM (little endian)`
  - `Target Architecture Variant`: `cortex-A5`
  - `Enable VFP extension support`: Enabled
  - `Target ABI`: `EABIhf`
  - `Floating point strategy`: `VFPv4-D16`
- `Toolchain`
  - `Toolchain type`: `External toolchain`
  - `Toolchain`: `Linaro ARM 2016.05`
  - `Toolchain origin`: `Toolchain to be downloaded and installed`
  - Select `Copy gdb server to the Target`
- `Target packages`
  - Keep `BusyBox` (default version) and keep the Busybox configuration proposed by Buildroot;
  - `Audio and video applications`
    - Select `alsa-utils`
    - `ALSA utils selection`
      - Select `alsactl`
      - Select `alsamixer`
      - Select `speaker-test`
    - Select `vorbis-tools`
- `Filesystem images`
  - Select `tar the root filesystem`

Exit the menuconfig interface. Your configuration has now been saved to the `.config` file.

Generate the embedded Linux system

Just run:

```
make
```

Buildroot will first create a small environment with the external toolchain, then download, extract, configure, compile and install each component of the embedded system.

All the compilation has taken place in the `output/` subdirectory. Let's explore its contents:

- `build`, is the directory in which each component built by Buildroot is extracted, and where the build actually takes place
- `host`, is the directory where Buildroot installs some components for the host. As Buildroot doesn't want to depend on too many things installed in the developer machines, it installs some tools needed to compile the packages for the target. In our case it installed *pkg-config* (since the version of the host may be ancient) and tools to generate the root filesystem image (*genext2fs*, *makedevs*, *fakeroot*).
- `images`, which contains the final images produced by Buildroot. In our case it's just a tarball of the filesystem, called `rootfs.tar`, but depending on the Buildroot configuration, there could also be a kernel image or a bootloader image.
- `staging`, which contains the "build" space of the target system. All the target libraries, with headers and documentation. It also contains the system headers and the C library, which in our case have been copied from the cross-compiling toolchain.
- `target`, is the target root filesystem. All applications and libraries, usually stripped, are installed in this directory. However, it cannot be used directly as the root filesystem, as all the device files are missing: it is not possible to create them without being root, and Buildroot has a policy of not running anything as root.

Run the generated system

Go back to the `$HOME/embedded-linux-labs/buildroot/` directory. Create a new `nfsroot` directory that is going to hold our system, exported over NFS. Go into this directory, and untar the rootfs using:

```
sudo tar xvf ../buildroot-2016.11.1/output/images/rootfs.tar
```

Add our `nfsroot` directory to the list of directories exported by NFS in `/etc/exports`, and make sure the board uses it too.

Boot the board, and log in (`root` account, no password).

You should now have a shell, where you will be able to run `speaker-test` and `ogg123` like you used to in the previous lab.

Going further

- Add tools to handle ubifs (Target packages -> Filesystem and Flash utilities -> mtd, jffs and ubi)
- Add dropbear (SSH server and client) to the list of packages built by Buildroot and log to your target system using an ssh client on your development workstation. Hint: you will have to set a non-empty password for the root account on your target for this to work.

- Flash the new system on the flash of the board
  - First, in buildroot, select the UBIFS filesystem image type.
  - You'll also need to provide buildroot some information on the underlying device that will store the filesystem. In our case, the logical eraseblock size is 124KiB (0x1f000), the minimum I/O unit size is 2048 (0x800) and the Maximum logical eraseblock (LEB) count is 1000.
  - Then, once the image has been generated, update your rootfs volume. From the U-Boot:

    ```
    mtdparts
    ubi part UBI
    tftp 0x22000000 rootfs.ubifs
    ubi writevol 0x22000000 rootfs ${filesize}
    ```

  - Boot the new system on the target and login via serial console. Then configure the network interface eth0 with a static ip address in the file `/etc/network/interfaces`:

    ```
    ...
    auto eth0
    iface eth0 inet static
       address 192.168.0.100
       netmask 255.255.255.0

    auto eth1
    iface eth1 inet dhcp
    ```

  - On the target set the root password using the command `passwd`.
  - Now try to login via ssh from the host:

    ```
    ssh root@192.168.0.100
    ```

- Add a new package in Buildroot for the GNU Gtypist game. Read the Buildroot documentation to see how to add a new package. Finally, add this package to your target system, compile it and run it. The newest versions require a library that is not fully supported by Buildroot, so you'd better stick with the latest version in the 2.8 series.

