Embedded Linux Training

Lab Book

Endocode AG
https://endocode.com

January 20, 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

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`.

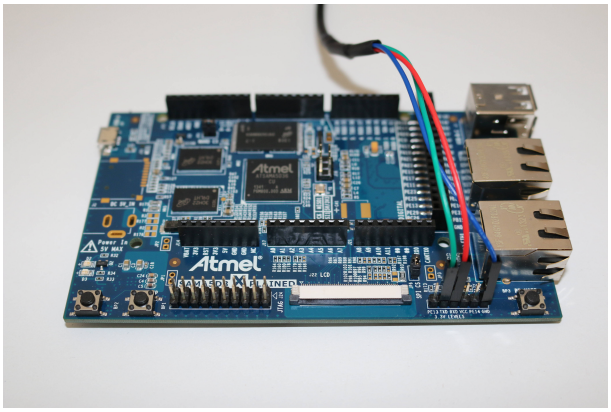
We first need to download this tool, from Atmel's website[1].

```
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.

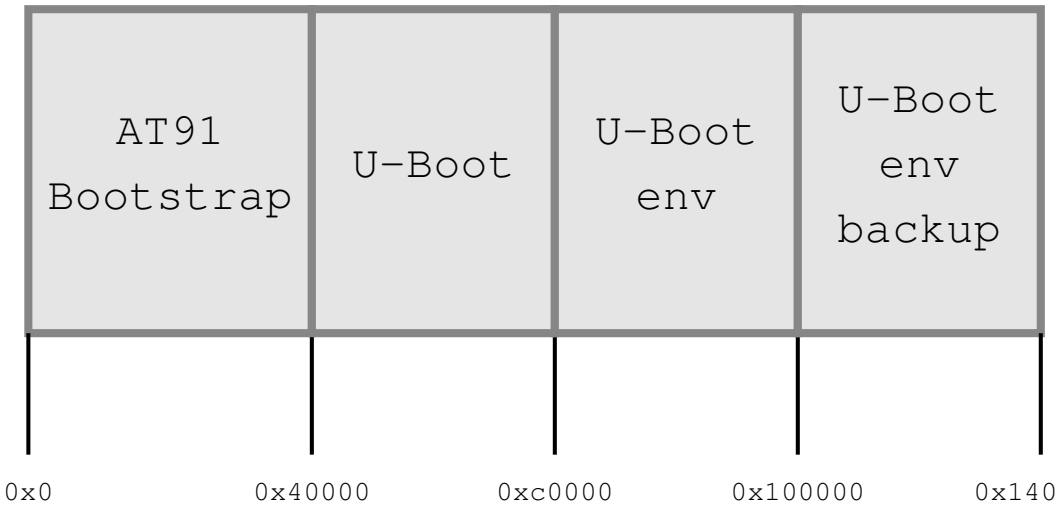
You can now power-up the board by connecting the micro-USB cable to the board, and to your PC at the other end. If a system was previously installed on the board, you should be able to interact with it through the serial line.

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.

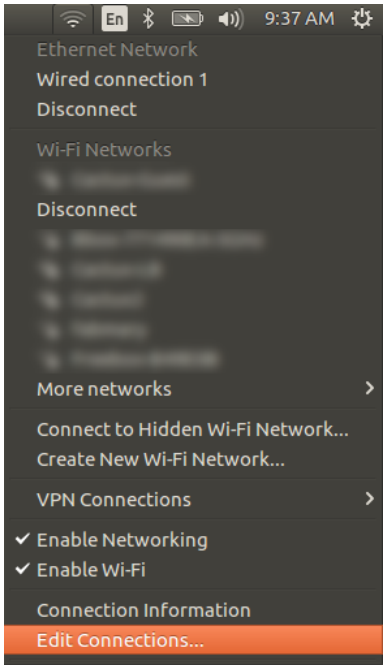
```
# Install the tftpd server package
sudo apt-get install tftpd-hpa
```



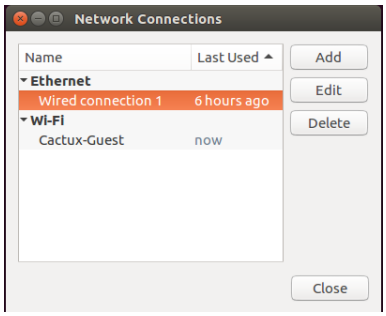
```
# Change the owner of '/var/lib/tftpboot' to allow the developer
# writing to the directory
sudo chown $USER /var/lib/tftpboot
```

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

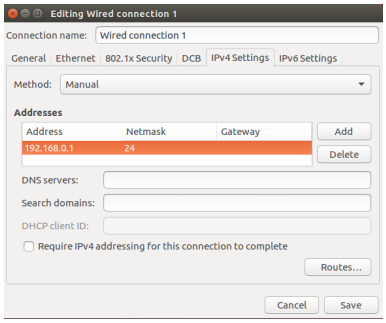
To configure this network interface on the workstation side, click on the *Network Manager* tasklet on your desktop, and select *Edit Connections*.



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.

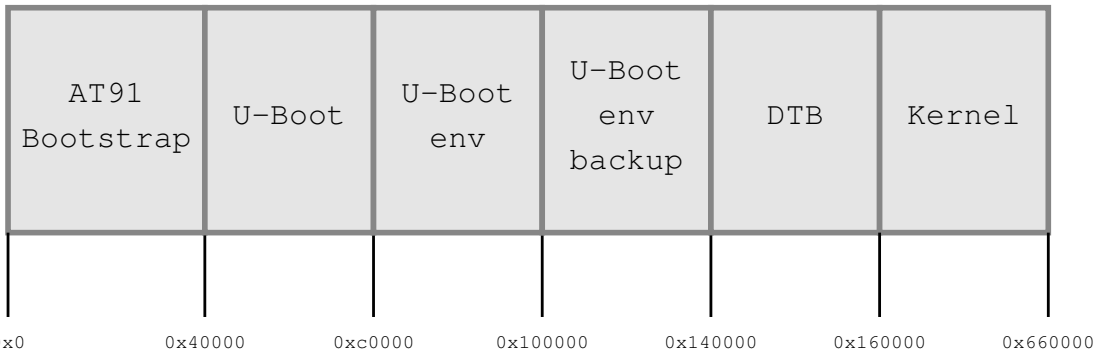
You can now automate all this every time the board is booted or reset. Reset the board, and specify a different `bootcmd`:

```
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:

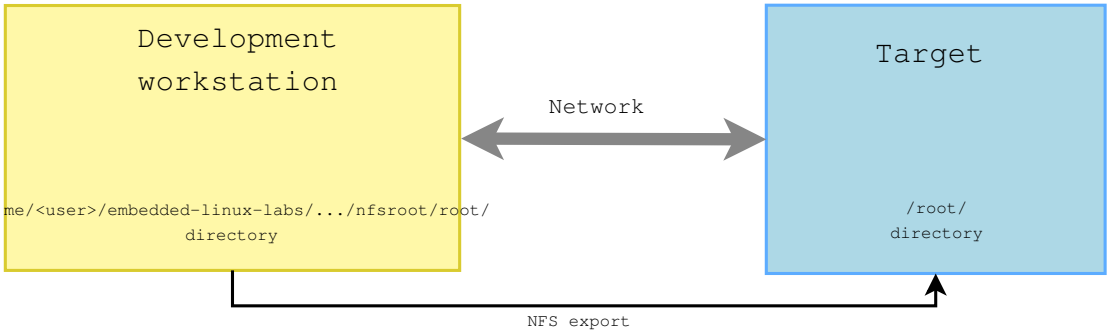
- be able to configure and build a Linux kernel that boots on a directory on your workstation, shared through the network by NFS.
- be able to create and configure a minimalistic root filesystem from scratch (ex nihilo, out of nothing, entirely hand made...) for the Xplained board
- understand how small 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

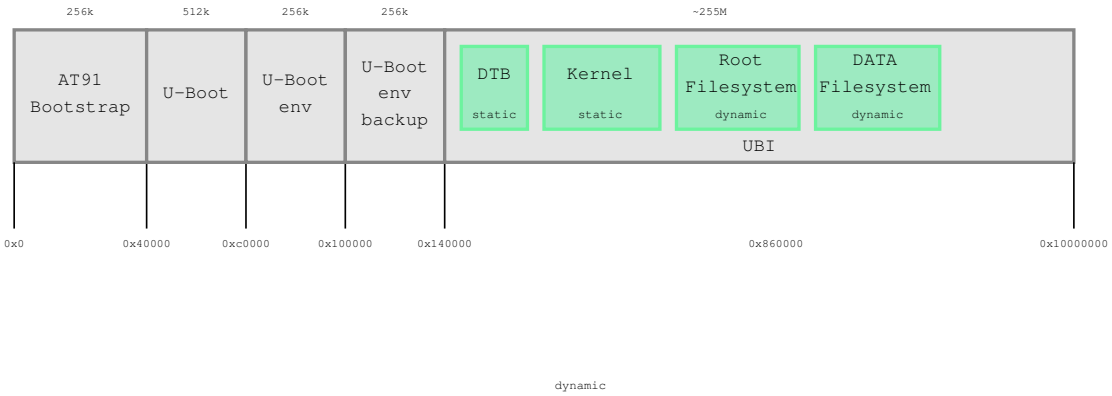
Instead of using an external MMC card as in the previous lab, we will make our system use its internal flash storage.

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
```

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>
```

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

```
setenv mtdparts mtdparts=<mtdid>:<size>(name),[<size>(name)]...
```

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.

Measuring boot time

Measuring the various components of boot time

Our first goal is to measure boot time in a coarse way. In a second step, we will measure the time spent in the various components of boot time.

Setting your goals

As explained in the lectures, the first thing to do is to choose what do measure. In the image running on this board, we choose to measure the time taken to start the demo application which lits up the red led.

Ideally, we would measure the time elapsed since power-on or since a reset event. However, we have no oscilloscope here, and therefore we can only rely on messages sent on the serial port.

- The earliest message we can have on the serial port is the `RomBOOT` one. We will take its time of arrival as the origin of time.
- We will implement a way to send a message to the serial line when the critical application starts.

Measuring overall boot time

We will use `grabserial` to measure the time spent while booting. It prints the timing information of each message received on the serial line[11], and display the time elapsed since a given event (like receiving a special string), used as the origin of time.

You will find `grabserial` on http://elinux.org/Grabserial.

Download it and put it in your `PATH`:

```
cd
mkdir bin && cd bin
wget https://raw.githubusercontent.com/tbird20d/grabserial/v1.9.3/grabserial
chmod a+x grabserial
```

The command we will be using is:

```
~/bin/grabserial -d /dev/ttyUSB0 -m RomBOOT -t -e 30
```

Here are details about this command:

- `-m "RomBOOT"` specifies the string marking the origin of time.
- `-t` makes `grabserial` show the time when **the first character of each line is received**.
- `-e 30` tells `grabserial` to stop measuring after 30 seconds.

[11] Beware that grabserial displays the arrival time of the beginning of a line on a serial port. If a message was sent without a trailing line feed, you could get the impression that the following characters have been received much earlier. This is particularly true when only a whitespace was first sent to the serial line.

First, you will have to exit `picocom` using `C-a C-x` to run `grabserial`.

Start `grabserial` and reset the board using `RESET`. You will see that it takes around 14 seconds to reach the Buildroot prompt:

```
$ ~/bin/grabserial -d /dev/ttyUSB0 -m RomBOOT  -t -e 30
[0.000001 0.000001] RomBOOT
[0.037473 0.037473]
[0.037561 0.000088]
[0.037610 0.000049] AT91Bootstrap 3.8.6 (Di 17. Jan 19:59:55 CET 2017)
[0.038939 0.001329]
[0.064419 0.025480] NAND: ONFI flash detected
[0.065132 0.000713] NAND: Manufacturer ID: 0x2c Chip ID: 0x32

...

[14.532369 0.014910] done
[14.558521 0.026152] Initializing random number generator... done.
[14.609159 0.050638] Starting network: OK
[14.858872 0.249713]
[14.862958 0.004086] Welcome to Buildroot
[14.863324 0.000366] buildroot login:
$
```

Note that there's no information on exactly when the first user space code gets executed. We will fix that in the *Add instrumentation* section.

You can make a visual check that the `buildroot login:` message is issued at about the same time as the red led lids up. Of course, this is not very accurate and it's time to measure the application starting time in a more accurate way.

Trick to estimate the application start time

For this kind of need, finding out when a device starts functioning, the ideal solution would be to monitor the pin with an oscilloscope, and based on the recorded activity, find out when the application actually starts.

When only software tracing is available, a standard solution is to add instrumentation to the application to write a message to the serial line.

In our case, however, the application is very simple and we could add such a message. In a complex piece of software probably without available source code this method is hard. We can write a message on the serial line before starting the application, but the time to load this application and start working is one of the components of boot time that we want to measure.

Reconstitute the build environment for the root filesystem

In every boot time reduction project, you have to rebuild the root filesystem, either by using the original build system, or by compiling the software stack with a new build system which could make it easier to optimize and simplify the root filesystem.

Another option would have been to make direct changes to the root filesystem. However, what you do is difficult to reproduce, and would make the system more difficult to maintain. In a real production project, it's much better to make changes to the configuration of the build system used to generate the root filesystem.

This way, your changes to reduce boot time are always applied, whatever the other changes you make on the system.

This is also true when you need to add instrumentation. If you do this by hand, you will lose all your instrumentation the next time you have to update or optimize a component.

Study how user space boots

It's important to understand how the various user space programs are started in the system. They all originate from the `init` program, which is the first and only program executed by the Linux kernel[12], when it's done booting the device.

The programs that are executed by the `init` program can found by reading the `/etc/inittab` file, which is a configuration file for `init`.

Start `picocom` if needed and study this file. You will see that a `/etc/init.d/rcS` script is also executed, and executes more scripts at his turn. Take some time to read these scripts, until you understand how the video player is started.

Add instrumentation

Make sure you are in the `boot-time-labs/buildroot/buildroot-2016.11.1/` directory.

Now, modify the `package/critical_application/S99critical_application` file[13]:

- Add an `echo -e "\nStarting critical application"` command [14] right before invoking `/usr/bin/app_launcher.sh`.
- Add an `echo -e "\nDone critical application"` right after the application.

The next thing we need is to know when the `init` program starts to execute.

Let's have a look at home this program is built. Run the below command to check the configuration of `BusyBox`, which implements `init`:

```
make busybox-menuconfig
```

Go to the `Init utilities` menu, and disable `Be _extra_ quiet on boot`. Exit and save your new configuration. This way, we will have a message on the serial line when `init` starts to execute.

It's now time to rebuild your root filesystem:

```
make
```

The above command should only take a few seconds to run.

Make sure that the `output/target/etc/init.d/S99critical_application` file contains the right modifications and that there are no syntax errors.

[12] There is one exception: the kernel can also run helpers to handle hotplug events.

[13] This is the initscript used to start the application.

[14] We are using a newline (`\n`) at the beginning of the string because `grabserial` only pays attention to the arrival time of the first character in each line. Without this, we would get the wrong time if the current line already had some characters sent earlier. If we didn't use `echo -e` (`e`: suppport for escape characters), `\n` wouldn't have been considered as a newline.

Re-flash the root filesystem

We will use `U-Boot` to update the root filesystem.

Let's copy our new root filesystem to it to our `tftpd` directory:

```
cp output/images/rootfs.ubi /var/lib/tftpboot/
```

Reboot your board into U-Boot and add a script to download the rootfs.ubi via tftp and write it to the mtd partition.

```
mtdparts
nand erase.part UBI
tftp 0x22000000 rootfs.ubi
nand write.trimffs 0x22000000 UBI ${filesize}
```

Press the reset button and make sure that your board boots as expected. You can also check that the new instrumentation messages are there. Don't pay attention to timing this time, as SSH keys had to be generated from scratch again.

Measure boot time components

You can now exit `picocom` and run `grabserial` again.

We can now time all the various steps of system booting in a pretty accurate way. Do it by filling writing down the time stamps in the below table.

Then, in each row, compute the elapsed time, which is the difference between the time stamp for the next step and the time stamp for the current step. For this purpose, you can use the `bc -l` command line calculator.

Boot phase	Start string	Time stamp	Elapsed
RomBOOT	`RomBOOT`		
Bootstrap	`AT91Bootstrap 3.8.6`		
Bootloader	`U-Boot 2016.05`		
Kernel	`Booting Linux on physical CPU 0`		
Init scripts	`init started: BusyBox v1.25.1`		
Critical application	`Starting critical application`		
Critical application ready	`Done critical application`		
Total			

In the next labs, we will work on reducing boot time for each of the boot phases.

Init script optimizations

Analysing and optimizing init scripts

Measuring

Remember that the first step in optimization work is measuring elapsed time. We need to know which parts of the init scripts are the biggest time consumers.

Use bootchartd on the board

Add `bootchartd` to your `BusyBox` configuration and also activate `FEATURE_SEAMLESS_GZ` [15].

```
make busybox-menuconfig
make
```

The above command should only take a few seconds to run.

Re-flash the root filesystem

Reflash your root filesystem in the same way you did it in the previous labs. Reboot it as usual to check if everything works.

The next thing to do is to use the `init` argument on the kernel command line (in `u-boot`, this is the `bootargs` environment variable) to boot using `bootchart` instead of using the `init` program provided by Busybox.

To go to the `u-boot` prompt, go to the `picocom` window showing the boards's serial line, reset your board with `RESET`, and press a key before the timer expires.

First, have a look at the current value of the `bootargs_base` environment variable:

```
U-Boot> printenv bootargs_base
bootargs_base=console=ttyS0,115200 rootfstype=ubifs ubi.mtd=4 ro root=ubi0:rootfs
```

Now add the `init` kernel parameter as follows:

```
U-Boot> setenv bootargs_base ${bootargs_base} init=/sbin/bootchartd
U-Boot> saveenv
U-Boot> boot
```

This will make the system boot and the resulting bootlog will be located in `/var/log/bootlog.tgz`. Copy that file on your host. You can do this by inserting an USB drive into one of the USB hosts ports of the board:

[15] Hint: to find where `bootchartd` support can be added in the `BusyBox` configuration interface, press the `/` key (search command). This will tell you in which submenu the corresponding option can be found.


```
# mount /dev/sda1 /mnt
# cp /var/log/bootlog.tgz /mnt
# umount /mnt
```

Copy the bootlog.tgz into your lab directory `~/embedded-linux-labs/boottime` on your host.

Analyse bootchart data on your workstation

To use `bootchart` on your workstation, you first need to install a few Java packages:

```
sudo apt-get install ant default-jdk
```

Now, get the `bootchart` source code[16] [17] , compile it and use `bootchart` to generate the boot chart:

```
cd ~/embedded-linux-labs/boottime
tar xf bootchart-0.9.tar.bz2
cd bootchart-0.9
ant
java -jar bootchart.jar ../bootlog.tgz
```

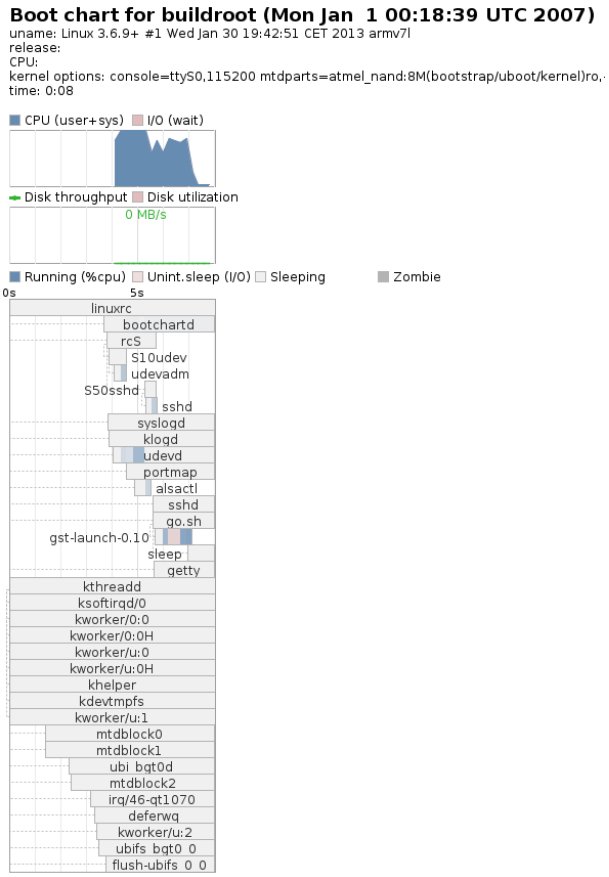
This produces the `bootlog.png` image which you can visualize to study and optimize your startup sequence:

```
xdg-open bootlog.png
```

`xdg-open` is a universal way of opening a file with a given MIME type with the associated application as registered in the system. According to the exact Ubuntu flavor that you are using (Ubuntu, Xubuntu, Kubuntu...), it will run the particular image viewer available in that particular flavor.

[16] The source code was originally found on http://prdownloads.sourceforge.net/bootchart/bootchart-0.9.tar.bz2.

[17] Don't try to get the `bootchart` package supplied by Ubuntu instead. While it has similar functionality, it looks like a completely unrelated piece of software. To confirm this, it has no dependency whatsoever on Java packages.



Remove unnecessary functionality

The above graph shows that there are `udev` processes running during the startup process, taking a significant amount of CPU time (the graph shows blue rectanges when the processes actually uses the CPU).

`udev` is pretty likely to be unnecessary in this system. Managing device files (even for hotplugged devices) is perfectly taken care of by the kernel's `devtmpfs` filesystem. `udev` might be needed to run specific programs to react to hotplugging events, but it has no visible usefulness in the basic demo usage scenario.

So let's remove `udev`! To do so, disable it in the Buildroot configuration:

```
make menuconfig
```

In `System configuration` and then in `/dev management`, choose `Dynamic using devtmpfs only`.

The next step would be to re-run Buildroot. However, we're hitting one of the current limitations of Buildroot. To keep its design simple and efficient, Buildroot currently doesn't support removing software even if it has been removed from the configuration. The clean way *would* be to run `make clean` and then run `make` again.

This is usually fine, as Buildroot is pretty fast.

Let's also take this opportunity to disable `bootchartd` compilation. Do this through `make busybox-menuconfig`.

Rebuild your root filesystem (run `make`) and reflash the device once again. Also remove `init=/sbin/bootchartd` from the kernel command line.

After the first reboot (as always for SSH key generation), reboot your board through `grabserial` and measure the time stamp for the `Done critical application` message.

Postpone services

If we get back to the bootchart that we generated, we can see that there are several other services which might be needed for the device to have all its features, but which should be run after the critical application, instead of before it. One example is the SSH server.

Let's customize the `/etc/init.d/rcS` file which starts such services, in order to start the critical application first, and the other services later.

Find the code for `/etc/init.d/rcS` in your buildroot. Then, add the below lines after the first line:

```
/usr/bin/app_launcher.sh &
sleep 2
```

The first added line line corresponds to the contents of `/etc/init.d/S99critical_application`. We remove this init script, because we will continue to use the `S??*` wildcard to start the other services.

According to the measures we have already made, the delay of 2 seconds should be more than enough to let the critical application run. With this pessimistic delay, we make sure that the CPU will only be dedicated to the critical application.

Adjust than your buildroot environment to fix them:

- By coping our new `rcS` file to the target filesystem.
- By removing the code that creates `/etc/init.d/S99critical_application`

The last thing to do before running `make` is to remove the `output/target/etc/init.d/S99critical_application` file manually. Remember that Buildroot doesn't clean the target directory!

Now run `make` and make sure that `output/target/etc/init.d/rcS` has the new right contents.

Reflash your system, and reboot it through `grabserial` to measure the new total boot time[18].

Write down your results in the table at the end of this chapter.

While the total boot time is now smaller, you could still notice that the time to run the critical application is slightly longer. It's not surprising. You can guess that an application is sharing some libraries (mainly C library components) with the executables for the other services. Since the services are no longer started before itself, it must be the first one to load such libraries, which explains the longer time to start. In our very simple demo case we hardly notice this.

Optimize necessary functionality

It's now time to optimize the init script that runs the application, by making this script as simple as possible.

[18] Note that we are no longer bothered by SSH key generation, as it starts after our application is started. Therefore, one reboot after reflashing is now sufficient.

Simplify shell programming

First, let's make a simple, one-time experiment that we won't keep. Directly on the board, modify `/usr/bin/app_launcher.sh`, replacing

```
if $(ls -la /sys/class/leds/d3/brightness > /dev/null 2> /dev/null );
```

by

```
if [ -f /sys/class/leds/d3/brightness ];
```

This is a much simpler test, with no output to manage. It is even very likely that the test command is a shell built-in, meaning that there is no sub-process to spawn (very costly).

Run `halt` and then reboot the board with `grabserial`, and measure how much time you saved by simplifying this test. In our own tests, we saved 34 ms! This is not negligible at all when similar cases accumulate and you are aiming at booting in just a few seconds.

The lesson to learn is that the shell scripts should be programmed with very good care, to avoid complex constructs causing multiple children processes to be run while fewer would be sufficient.

Radical simplification of the launcher script

Now, let's make `/usr/bin/app_launcher.sh` as simple as possible:

- By removing the soc subtype check. This check was very complex, and completely unnecessary when you develop for exactly this SoC.

As `app_launcher.sh` comes from the upstream source package we need to patch it during build. Go into the `output/build/` directory and create a backup of the source directory of critical_application.

```
cp -a critical_application-0.0.3 critical-application-0.0.3.BACK
```

Now simplify the script by removing lines. Also simplify the check as discussed above. Save the file and create a patch from the backed up copy and the changed file by running:

```
diff -u critical_application-0.0.3/app_launcher.sh \
        critical_application-0.0.3.BACK/app_launcher.sh  \
        >  simplify_app_launcher.patch
```

Copy the patch into the package directory `package/critical_application/`.

As usual, update your root filesystem, reflash it, measure the new results and store them in the table below.

Going further

If you are ahead of the others in the workshop, there are other thing you could try to do to start the application earlier: You could

- Take the commands that are run before `/etc/init.d/rcS` in `/etc/inittab`, and move them after starting the application in `/etc/init.d/rcS`.
- In `/etc/init.d/rcS`, source `/usr/bin/app_launcher.sh` instead of executing it. This way, this shell script will be interpreted by the current shell, instead of having to spawn a child process (another shell, this is expensive).

Last but not least, there's another thing we can simplify. We could recompile `BusyBox` with only the capabilities that are needed in this demo. It would make the `init` program and the other executables faster to load (being smaller) and run.

However, we prefer to make such a simplification at the end, because it is convenient to interact with a full-featured set of UNIX commands when you experiment with your system.

Results

Fill the below table with the results from your experiments:

Technique	Total time stamp	Difference with previous experiment
Original time		N/A
Remove `udev`		
Postpone services		
Optimize launcher		
Other trick		
Other trick		
Total gain		

Application optimization

Optimize the startup time of your applications

Measuring

The general rule stays the same. You have to measure the time taken to execute the various pieces of code in your application.

Here, except for possible compiler optimization, modifying the application is outside the scope of this workshop, because it requires a good knowledge about the application itself.

However, we are going to use a few techniques which should help you to improve your own application when you are back to real life.

Compiling utilities

With a build system like Buildroot, it's easy to add performance analysis and debugging utilities.

Configure Buildroot to add `strace` to your root filesystem. You will find the corresponding configuration option in `Package selection for the target` and then in `Debugging, profiling and benchmark`.

Run Buildroot and reflash your device as usual.

Tracing and profiling with strace

With `strace`'s help, you can already have a pretty good understanding of how your application spends it time. You can see all the system calls that it makes and knowing the application, you can guess in which part of the code it is at a given time.

You can also spot unnecessary attempts to open files that do not exist, multiple accesses to the same file, or more generally things that the program was not supposed to do. All these correspond to opportunities to fix and optimize your application.

Once the board has booted, run `strace` on the app_launcher.sh application:

```
strace -tt -f -o strace.log /usr/bin/app_launcher.sh
```

Also have `strace` generate a summary:

```
strace -c -f -o strace-summary.log /usr/bin/app_laucher.sh
```

Take some time to read `strace.log`[19]

[19] At this stage, when you have to open files directly on the board, some familiarity with the basic commands of the `vi` editor becomes useful. See `http://free-electrons.com/doc/command_memento.pdf` for a basic command summary. Otherwise, you can use the more rudimentary `more` command. You can also copy the files to your PC, using a USB drive, for example.

Also have a look at `strace-summary.log`. You will find the number of errors trying to open files that do not exist, for example. You can also count the number of memory allocations (using the `mmap2` system call).

Removing unnecessary functionality

Based on what you learned by tracing and profiling your application, you could recompile it to remove support for the features that you know are not used in your system. This should speed up its execution, at least slightly.

Postponing, reordering

Modifying a complex application would be very difficult. It could make sense with your own application though, for which the code is familiar to you.

Kernel optimizations

Measure kernel boot components and optimize the kernel boot time

Measuring

We are going to use the kernel `initcall_debug` functionality.

The first thing we need to do is to recompile the kernel to make sure it has the right options.

We use our prepared buildroot to rebuild the kernel.

To configure the kernel with the below settings:

- `CONFIG_LOG_BUF=16`. That's the size of the kernel ring buffer. If we keep the default size (`14`), the earliest kernel messages are overwritten by the latest ones.
- `CONFIG_PRINTK_TIME=y`

Run `make menuconfig`, go to the `Kernel` menu and add the prepared kernel config snippet as `Additional configuration fragment files`

Save your configuration and run `make` to rebuild your image.

Reflash your system and make it boots in the same way as before.

Now, let's enable `initcall_debug`. Reboot your board and press a key to stop the U-boot countdown.

In the U-boot command line, add settings to the kernel command line[20] and boot your system:

```
setenv bootargs_base "${bootargs_base} initcall_debug printk.time=1"
boot
```

Once booted, you can store the debug information in a file:

```
dmesg > initcall_debug.log
```

Copy this file to your host into the `boottime/` lab directory.

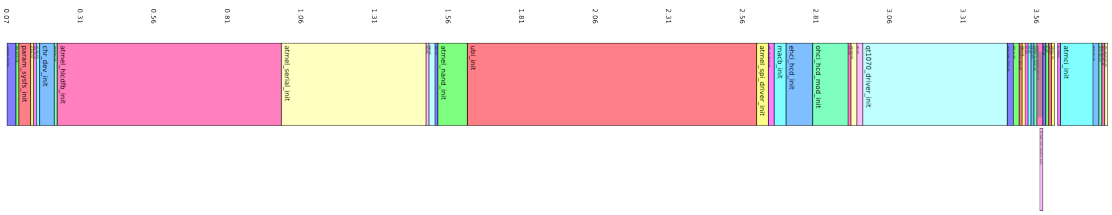
Let's use the extracted kernel sources in `output/build/linux-4.9.4` to run a script to generate a nice boot graph from this debug information:

```
perl linux-4.9.4/scripts/bootgraph.pl initcall_debug.log > boot.svg
```

You can view the boot graph with the `inkscape` vector graphics editor:

```
sudo apt-get install inkscape
inkscape boot.svg
```

[20] Don't save these settings with `saveenv`. We will just need them once.



Now review the longest initcalls in detail. Each label is the name of a function in the kernel sources. Try to find out in which source file each function is defined[21], and what each driver corresponds to.

Then, you can look the source code and try look for obvious causes which would explain the very long execution time: delay loops (look for `delay`, parameters which can reduce probe time but are not used, etc).

Before going on, reboot your board through `grabserial` to measure the total boot time. The kernel rebuild could have modified it a little bit. Write down your result at the end of this chapter.

Reordering and postponing functionality

The boot graph revealed drivers which could be compiled as modules and loaded later. Let's compile such drivers that you found as modules and install the corresponding modules in the root filesystem.

Because of this requirement, we now need to compile the kernel with Buildroot.

First let's prepare a kernel configuration in which the selected drivers are now compiled as modules:

```
tar -xf linux-4.9.4.tar.xz
cd linux-4.9.4
export ARCH=arm
make sama5_defconfig
make menuconfig
```

For each driver, you have to look for the parameter which enables it. If the parameter name is not trivial to find in the kernel configuration interface, already knowing the source file implementing the function in the boot chart, you can look at the `Makefile` file in the same directory, and find which parameter is used to compile the source file in a conditional way.

Once you are done converting static drivers into modules, copy your configuration file:

```
cp .config ../config-4.9.4-with-modules
```

Now go back to the Buildroot directory and run:

```
make menuconfig
```

Go to the `Kernel` menu and fill the kernel related settings:

- `Kernel version`: `Custom version`
- `(4.9.4) Kernel version`
- `Kernel configuration`: `Using a custom (def)config file`

[21] You can do it with utilities such as `cscope` or through our on-line service to explore the Linux kernel sources: http://lxr.free-electrons.com

- `Configuration file path: ../config-4.9.4-with-modules`
- Enable `Device tree support`
- `Device tree source: Use a device tree present in the kernel`
- `Device Tree Source file names: at91-sama5d3_xplained`

Check that your modules have been added to the root filesystem:

```
find output/target -name *.ko
```

The last thing to do is to load the necessary modules in a manual way[22] from the `/etc/init.d/rcS` file, before starting the services (SSH will need the network driver to be loaded, for example).

Modify the `rcS` file to run the `modprobe` command for each of the modules before starting the services (example: `modprobe macb`).

Rerun Buildroot. Reflash your device, measure the new boot time and write it down at the end of this chapter.

Removing unnecessary functionality

The boot graph that we generated doesn't show any obvious kernel driver that would consume a significant amount of time and could be taken away because it is completely useless.

Of course, there will be kernel features that we will be able to remove, in order to reduce the kernel size and make the kernel faster to load in the bootloader. However, this shouldn't have much impact on the kernel's execution time.

There's one thing we can remove though, and didn't appear on the boot graph: we can disable console output. Writing messages on the serial line can be very slow, especially as the serial line has a slow bandwidth.

You can do this by adding the `quiet` parameter to the kernel command line.

Look for the `bootargs_base` and add the `quiet` parameter.

Restart your device, measure boot time, and write in down in the summary table.

There is another thing that is unnecessary too: the calibration of the delay loop, as explained in the lectures. Read the `lpj` value from a previous boot log, and pass this value on the kernel command line.

Measure the new boot time and write your result in the summary table.

Optimizing necessary functionality

The boot graph revealed the existence of drivers with initcalls taking a long time to execute. It would be worth spending time analysing their code, looking for opportunities to reduce the initialization time taken by these drivers.

However, such investigation work could take days, unless you find obvious issues (such as big delay loops).

[22] Oops, we don't have `udev` any more, and it would have done it for us. However, manually loading modules is no big deal, and a simpler solution for hotplug events can still be put in place anyway.

Results

Fill the below table with the results from your experiments:

Technique	Total time stamp	Difference with previous experiment
Original time		N/A
Postpone functionality (drivers compiled as modules)		
Remove unnecessary functionality (`quiet` option)		
Remove unnecessary functionality (skip delay loop calibration)		
Total gain		

Real-time - Timers and scheduling latency

Objective: Learn how to handle real-time processes and practice with the different real-time modes. Measure scheduling latency.

After this lab, you will:

- Be able to check clock accuracy.
- Be able to start processes with real-time priority.
- Be able to build a real-time application against the standard POSIX real-time API, and against Xenomai's POSIX skin.
- Have compared scheduling latency on your system, between a standard kernel, a kernel with `PREEMPT_RT` and a kernel with Xenomai.

Setup

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

Install the `netcat` package.

Root filesystem

To compare real-time latency between standard Linux and Xenomai, we are going to need a root filesystem and a build environment that supports Xenomai.

Let's build this with Buildroot.

Reuse and extract the Buildroot 2016.11.1 sources in the `realtime` directory. Configure Buildroot by copying the prepared config from the `extras/` directory to the buildroot directory.

Check if the config has the following settings already enabled:

- In `System configuration`:
  - in `Run a getty (login prompt) after boot`, `TTY port: ttyS0`
- In `Target packages`:
  - Enable `Show packages that are also provided by busybox`. We need this to build the standard `netcat` command, not provided in the default BusyBox configuration.
  - In `Debugging, profiling and benchmark`, enable `rt-tests`. This will be a few applications to test real-time latency.
  - In `Networking applications`, enable `netcat`
  - In `Real-Time`, enable `Xenomai Userspace`:
    * Enable `Install testsuite`

  * Make sure that `POSIX skin library` and `Native skin library`[23] are enabled.

Now, build your root filesystem.

After the build finished copy the files in `realtime/extras/target/` to `buildroot-2016.11.1/output/target/root/`. Run `make` again to add the files to the image. At the end of the build job flash the target as usual.

Compiling with the POSIX RT library

The root filesystem was built with the GNU C library, because it has better support for the POSIX RT API.

In our case, when we created this lab, uClibc didn't support the `clock_nanosleep` function used in our `rttest.c` program. *uClibc* also does not support priority inheritance on mutexes.

Therefore, we will need to compile our test application with the toolchain that Buildroot used.

Let's configure our `PATH` to use this toolchain:

```
export PATH=$HOME/embedded-linux-labs/realtime/buildroot-2016.11.1/output/host/usr/bin:$PATH
```

Have a look at the `rttest.c` source file available in the `realtime/extras/target` directory. See how it shows the resolution of the `CLOCK_MONOTONIC` clock.

Now compile this program:

```
arm-linux-gnueabi-gcc -o rttest rttest.c -lrt
```

Execute the program on the board. Is the clock resolution good or bad? Compare it to the timer tick of your system, as defined by `CONFIG_HZ`. (You also can compile the `rttest.c` for your host.)

Copy the results in a file, in order to be able to compare them with further results.

Obviously, this resolution will not provide accurate sleep times, and this is because our kernel doesn't use high-resolution timers. So let's enable the `CONFIG_HIGH_RES_TIMERS` option in the kernel configuration.

Recompile your kernel using the prepared config file `realtime/extras/linux-4.9.high_res_timers`, flash and boot your target with the new version, and check the new resolution. Better, isn't it?

Testing the non-preemptible kernel

Now, do the following tests:

- Test the program with nothing special and write down the results.
- Test your program and at the same time, add some workload to the board, by running `/root/doload 300 > /dev/null 2>&1 &` on the board, and using `netcat 192.168.0.100 5566` on your workstation in order to flood the network interface of the Xplained board (where `192.168.0.100` is the IP address of the Xplained board).
- Test your program again with the workload, but by running the program in the `SCHED_FIFO` scheduling class at priority `99`, using the `chrt` command.

[23] Needed by the Xenomai testsuite.

Testing the preemptible kernel

Recompile your kernel with `CONFIG_PREEMPT` enabled, which enables kernel preemption (except for critical sections protected by spinlocks).

Run the simple tests again with this new preemptible kernel and compare the results.

Compiling and testing the PREEMPT_RT kernel

In your buildroot directory run `make menuconfig` go to the `Kernel` menu, configure your kernel by using `../extras/linux-4.9.preempt_full` as `Configuration file path` and adding `../extras/patch-4.9-rt1.patch` as `Custom kernel patch`.

Build and flash the image and boot the target.

Repeat the tests and compare the results again. You should see a massive improvement in the maximum latency.

