Embedded Linux System Development

Microchip SAMA5D3 variant

Practical Labs

bootlin

https://bootlin.com

December 17, 2019

About this document

Updates to this document can be found on https://bootlin.com/doc/training/embedded-linux/.

This document was generated from LaTeX sources found on https://github.com/bootlin/training-materials.

More details about our training sessions can be found on https://bootlin.com/training.

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://bootlin.com/doc/training/embedded-linux/embedded-linux-labs.tar.xz
tar xvf embedded-linux-labs.tar.xz
```

Lab data are now available in an `embedded-linux-labs` directory in your home directory. This directory contains directories and files used in the various practical labs. It will also be used as working space, in particular to keep generated files separate when needed.

You are now ready to start the real practical labs!

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/`

Building a cross-compiling toolchain

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

After this lab, you will be able to:

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

Setup

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

Install needed packages

Install the packages needed for this lab:

```
sudo apt install build-essential git autoconf bison flex \
     texinfo help2man gawk libtool-bin libncurses5-dev
```

Getting Crosstool-ng

Let's download the sources of Crosstool-ng, through its git source repository, and switch to a commit that we have tested:

```
git clone https://github.com/crosstool-ng/crosstool-ng.git
cd crosstool-ng/
git checkout eb65ba65
```

Installing Crosstool-ng

We can either install Crosstool-ng globally on the system, or keep it locally in its download directory. We'll choose the latter solution. As documented at http://crosstool-ng.github.io/docs/install/#hackers-way, do:

```
./bootstrap
./configure --enable-local
make
```

Then you can get Crosstool-ng help by running

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

Configure the toolchain to produce

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

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

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

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

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

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

In `Path and misc options`:

- Change `Maximum log level to see` to `DEBUG` so that we can have more details on what happened during the build in case something went wrong.

In `Toolchain options`:

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

In `C-library`:

- Enable `IPv6` support. That's because of Buildroot (which we will use later, which doesn't accept to use toolchains without IPv6 support.

In `Debug facilities`, disable every option. Some of these options will be useful in a real toolchain, but in our labs, we will do debugging work with another toolchain anyway. Hence, not compiling debugging features here will reduce toolchain building time.

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

Produce the toolchain

Nothing is simpler:

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

The toolchain will be installed by default in `$HOME/x-tools/`. That's something you could have changed in Crosstool-ng's configuration.

And wait!

Known issues

Source archives not found on the Internet

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

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

Testing the toolchain

You can now test your toolchain by adding `$HOME/x-tools/arm-training-linux-uclibcgnueabihf/bin/` to your `PATH` environment variable and compiling the simple `hello.c` program in your main lab directory with `arm-linux-gcc`.

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

Cleaning up

Do this only if you have limited storage space. In case you made a mistake in the toolchain configuration, you may need to run Crosstool-ng again. Keeping generated files would save a significant amount of time.

To save about 7 GB of storage space, do a `./ct-ng clean` in the Crosstool-NG source directory. This will remove the source code of the different toolchain components, as well as all the generated files that are now useless since the toolchain has been installed in `$HOME/x-tools`.

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 Microchip'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`.

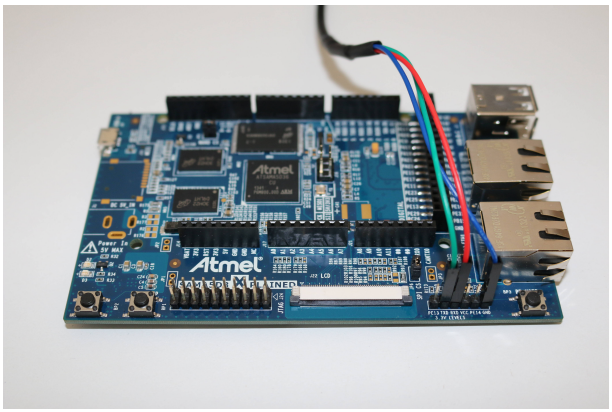
We first need to download this tool, from Microchip's website[1].

```
wget https://ww1.microchip.com/downloads/en/DeviceDoc/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 https://bootlin.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 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
```

Important: for the group change to be effective, in Ubuntu 18.04, you have to *completely reboot* the system [2]. A workaround is to run `newgrp dialout`, but it is not global. You have to run it in each terminal.

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.

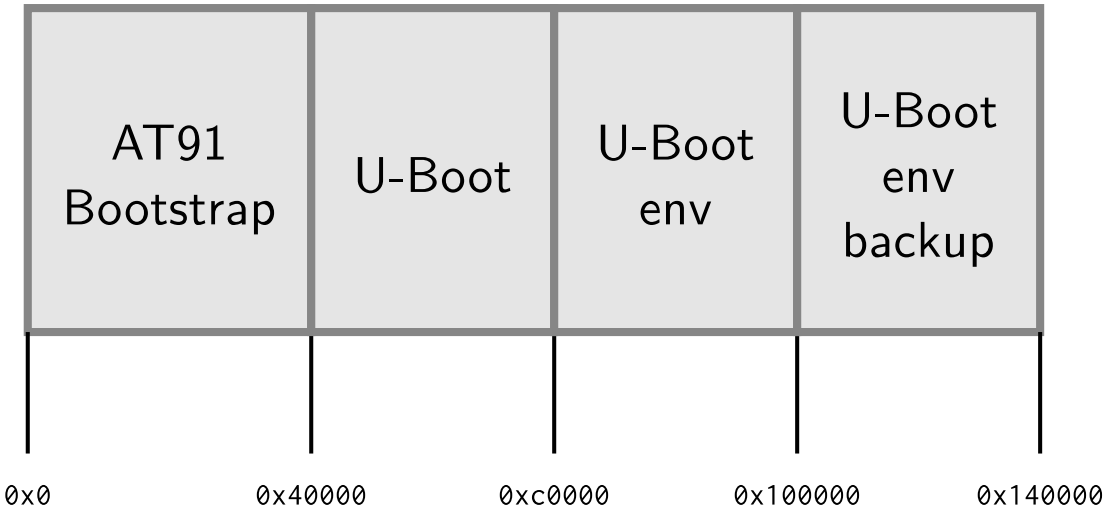
If you wish to exit `picocom`, press `[Ctrl][a]` followed by `[Ctrl][x]`.

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:

[2] As explained on https://askubuntu.com/questions/1045993/after-adding-a-group-logoutlogin-is-not-enough-in-18-04/.



- 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 Microchip's GitHub account:

```
git clone https://github.com/linux4sam/at91bootstrap.git
cd at91bootstrap
git checkout v3.8.9
```

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 specify the cross-compiler prefix (the part before `gcc` in the cross-compiler executable name):

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

Last but not least, install the `python` package that the Makefile for AT91Bootstrap will try to invoke.

You can now start compiling using `make`[3].

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_64`, running 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 [4] parameters to what RomBOOT expects.
- 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-2017.09.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.
- In recent versions of U-Boot and for some boards, you will need to have the Device Tree compiler:

  ```
  sudo apt install device-tree-compiler
  ```

- Finally, run `make`, which should build U-Boot.

[3] 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.

[4] *ECC* means *Error Correcting Code*. If we don't have the same ECC scheme as the one RomBOOT expects, RomBOOT will think that the NAND contents are corrupted.

Shrinking U-Boot

Look at the size of the `u-boot.bin` binary. According to the above flash layout, the U-Boot binary is supposed to fit between flash offset `0x40000` and offset `0xc0000`, corresponding to a maximum size of `524288` bytes. Is `u-boot.bin` bigger than this maximum?

The first offset is what AT91Bootstrap expects (though it can be changed in AT91Bootstrap's configuration). The second one, corresponding to where U-Boot stores its environment settings, is board dependent but apparently cannot be changed through `make menuconfig`.

To avoid recompiling AT91Bootstrap, we propose to compile U-Boot with less features, to make its binary small. That's probably something you will do too during real-life projects.

So, in U-Boot sources, run `make menuconfig`, look for and disable the below options:[5].

- `ext4` options
- `nfs` options
- `USB` options
- `SPL` options
- `XIMG` support
- `FIT` (Flattened Image Tree) support
- `CMD_ELF` option
- `dhcp` command support
- Regular expression support (`REGEX`)
- `loadb` support
- `CMD_MII` option

Now, recompile U-Boot and check that `u-boot.bin` is smaller than our maximum size.

Flashing U-Boot

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

You can now exit `sam-ba`.

Testing U-Boot

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

```
AT91Bootstrap 3.8.9 (Mon Oct 30 2017 16:09:08 (UTC+0100))

NAND: ONFI flash detected
NAND: Manufacturer ID: 0x2c Chip ID: 0xda
NAND: Page Bytes: 2048, Spare Bytes: 64
NAND: ECC Correctability Bits: 4, ECC Sector Bytes: 512
NAND: Disable On-Die ECC
```

[5] For each option, don't hesitate to use help information to find out what it is about


```
NAND: Initialize PMECC params, cap: 4, sector: 512
NAND: Image: Copy 0xa0000 bytes from 0x40000 to 0x26f00000
NAND: Done to load image
<debug_uart>

U-Boot 2017.09 (Oct 30 2017 - 16:20:29 +0100)

CPU: SAMA5D36
Crystal frequency:       12 MHz
CPU clock        :      528 MHz
Master clock     :      132 MHz
DRAM:  256 MiB
NAND:  256 MiB
MMC:   Atmel mci: 0, Atmel mci: 1
*** Warning - bad CRC, using default environment

In:    serial@ffffee00
Out:   serial@ffffee00
Err:   serial@ffffee00
Net:
Error: ethernet@f0028000 address not set.
No ethernet found.
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.

With a network cable, connect the Ethernet port labelled ETH0/GETH of your board to the one of your computer. If your computer already has a wired connection to the network, your instructor will provide you with a USB Ethernet adapter. A new network interface should appear on your Linux system.

Find the name of this interface by typing:

```
ifconfig -a
```

The network interface name is likely to be `enxxx`[6]. If you have a pluggable Ethernet device, it's easy to identify as it's the one that shows up after pluging in the device.

Then, instead of configuring the host IP address from NetWork Manager's graphical interface, let's do it through its command line interface, which is so much easier to use:

```
nmcli con add type ethernet ifname en... ip4 192.168.0.1/24
```

[6] Following the *Predictable Network Interface Names* convention: https://www.freedesktop.org/wiki/Software/systemd/PredictableNetworkInterfaceNames/

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[7].

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

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

The `tftp` command should have downloaded the `textfile.txt` file from your development workstation into the board's memory at location `0x22000000`[8].

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

```
md 0x22000000
```

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

Rescue binaries

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

[7] 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.

[8] 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://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11121-32-bit-Cortex-A5-Microcontroller-SAMA5D3_Datasheet_B.pdf. It's a big document (more than 1,900 pages). In this document, look for `Memory Mapping` and you will find the SoC memory map. You will see that the address range for the memory controller (*DDRC S*) starts at `0x20000000` and ends at `0x3fffffff`. This shows that the `0x22000000` address is within the address range for RAM. You can also try with other values in the same address range, knowing that our board only has 256 MB of RAM (that's `0x10000000`, so the physical RAM probably ends at `0x30000000`).

Kernel sources

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

After this lab, you will be able to:

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

Setup

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

Get the sources

Go to the Linux kernel web site (`https://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-4.19.x`, which this lab was tested with.

To practice with the `patch` command later, download the full 4.18 sources. Unpack the archive, which creates a `linux-4.18` directory. Remember that you can use `wget <URL>` on the command line to download files.

Apply patches

Download the patch files corresponding to the latest 4.19 stable release: a first patch to move from 4.18 to 4.19 and if one exists, a second patch to move from 4.19 to 4.19.x.

Without uncompressing them to a separate file, apply the patches to the Linux source directory.

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

Rename the `linux-4.18` directory to `linux-4.19.<x>`.

Kernel - Cross-compiling

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

After this lab, you will be able to:

- Set up a cross-compiling environment
- Configure the kernel Makefile accordingly
- Cross compile the kernel for the Microchip 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.

Target system

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

Kernel sources

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

Cross-compiling environment setup

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

```
export PATH=$HOME/x-tools/arm-training-linux-uclibcgnueabihf/bin:$PATH
```

Also, don't forget to either:

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

Linux kernel configuration

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

Once found, use this target to configure the kernel with the ready-made configuration.

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

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

Cross compiling

At this stage, you need to install the `libssl-dev` package to compile the kernel.

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

```
make
```

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

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

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

Load and boot the kernel using U-Boot

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

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

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

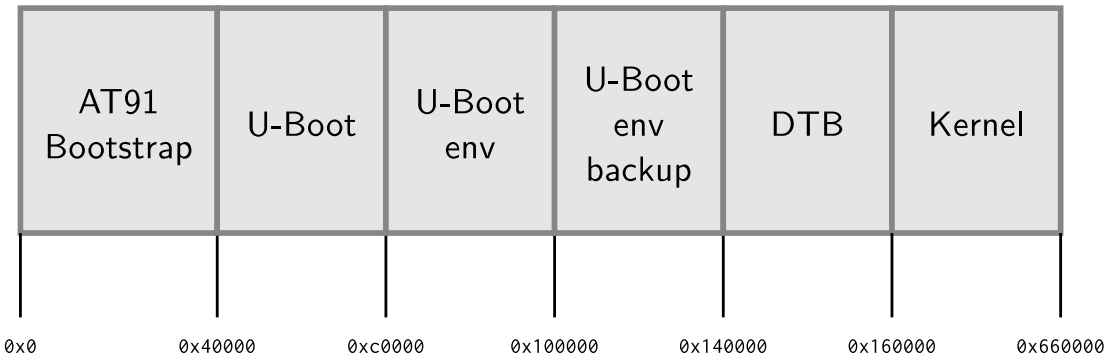
You can now automate all this every time the board is booted or reset. Reset the board, and specify a different `bootcmd`:

```
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[9]

[9] 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:

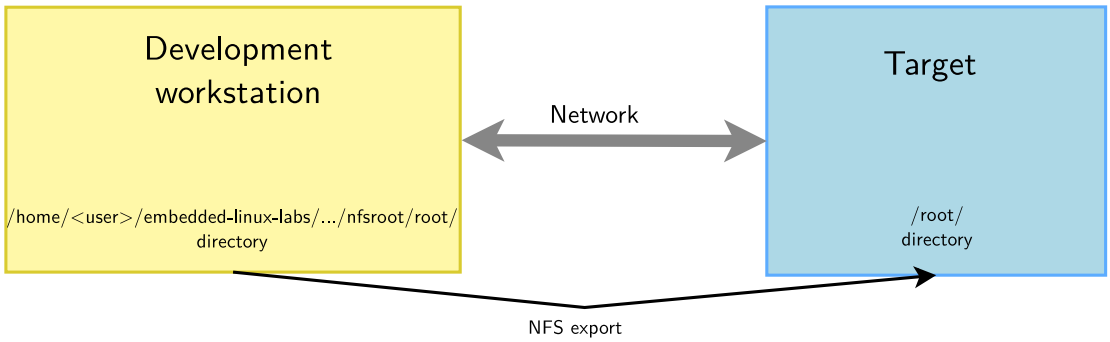
- be able to configure and build a Linux kernel that boots on a directory on your workstation, shared through the network by NFS.
- be able to create and configure a minimalistic root filesystem from scratch (ex nihilo, out of nothing, entirely hand made...) for the Xplained board.
- understand how small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

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)
```

Of course, replace `<user>` by your actual user name.

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 root=/dev/nfs ip=192.168.0.100:::::eth0
   nfsroot=192.168.0.1:/home/<user>/embedded-linux-labs/tinysystem/nfsroot,nfsvers=3 rw
```

Once again, replace `<user>` by your actual user name.

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 1.29.x release.

To configure BusyBox, we won't be able to use `make xconfig`, which is currently broken for BusyBox in Ubuntu 18.04, because it requires an old version of the Qt library.

So, let's use `make menuconfig`.

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[10]. 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[11].

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.

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

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

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.

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[12], 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.

[12] Invoke your cross-compiler in the same way you did during the toolchain lab

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 - Block file systems

Objective: configure and boot an embedded Linux system relying on block storage

After this lab, you will be able to:

- Manage partitions on block storage.
- Produce file system images.
- Configure the kernel to use these file systems
- Use the tmpfs file system to store temporary files

Goals

After doing the *A tiny embedded system* lab, we are going to copy the filesystem contents to the SD card. The filesystem will be split into several partitions, and your sama5d3 X-plained board will be booted with this SD card, without using NFS anymore.

Setup

Throughout this lab, we will continue to use the root filesystem we have created in the `$HOME/embedded-linux-labs/tinysystem/nfsroot` directory, which we will progressively adapt to use block filesystems.

Filesystem support in the kernel

Recompile your kernel with support for SquashFS and ext4[13].

Update your kernel image in NAND flash.

Boot your board with this new kernel and on the NFS filesystem you used in this previous lab.

Now, check the contents of `/proc/filesystems`. You should see that ext4 and SquashFS are now supported.

Prepare the SD card

We're going to use an SD card for our block device.

Plug the SD card your instructor gave you on your workstation. Type the `dmesg` command to see which device is used by your workstation. In case the device is `/dev/mmcblk0`, you will see something like

```
[46939.425299] mmc0: new high speed SDHC card at address 0007
[46939.427947] mmcblk0: mmc0:0007 SD16G 14.5 GiB
```

[13] Basic configuration options for these filesystems will be sufficient. No need for things like extended attributes.

The device file name may be different (such as `/dev/sdb` if the card reader is connected to a USB bus (either inside your PC or using a USB card reader).

In the following instructions, we will assume that your SD card is seen as `/dev/mmcblk0` by your PC workstation.

Type the `mount` command to check your currently mounted partitions. If SD partitions are mounted, unmount them:

```
$ sudo umount /dev/mmcblk0*
```

Then, clear possible SD card contents remaining from previous training sessions (only the first megabytes matter):

```
$ sudo dd if=/dev/zero of=/dev/mmcblk0 bs=1M count=256
```

Now, let's use the `cfdisk` command to create the partitions that we are going to use:

```
$ sudo cfdisk /dev/mmcblk0
```

If `cfdisk` asks you to `Select a label type`, choose `dos`. This corresponds to traditional partitions tables that DOS/Windows would understand. `gpt` partition tables are needed for disks bigger than 2 TB.

In the `cfdisk` interface, delete existing partitions, then create three primary partitions, starting from the beginning, with the following properties:

- One partition, 64MB big, with the `FAT16` partition type.
- One partition, 8 MB big[14], that will be used for the root filesystem. Due to the geometry of the device, the partition might be larger than 8 MB, but it does not matter. Keep the `Linux` type for the partition.
- One partition, that fills the rest of the SD card, that will be used for the data filesystem. Here also, keep the `Linux` type for the partition.

Press `Write` when you are done.

To make sure that partition definitions are reloaded on your workstation, remove the SD card and insert it again.

Data partition on the SD card

Using the `mkfs.ext4` create a journaled file system on the third partition of the SD card:

```
sudo mkfs.ext4 -L data -E nodiscard /dev/mmcblk0p3
```

- `-L` assigns a volume name to the partition
- `-E nodiscard` disables bad block discarding. While this should be a useful option for cards with bad blocks, skipping this step saves long minutes in SD cards.

Now, mount this new partition and move the contents of the `/www/upload/files` directory (in your target root filesystem) into it. The goal is to use the third partition of the SD card as the storage for the uploaded images.

Connect the SD card to your board. You should see the partitions in `/proc/partitions`.

[14] For the needs of our system, the partition could even be much smaller, and 1 MB would be enough. However, with the 8 GB SD cards that we use in our labs, 8 MB will be the smallest partition that `cfdisk` will allow you to create.

Mount this third partition on `/www/upload/files`.

Once this works, modify the startup scripts in your root filesystem to do it automatically at boot time.

Reboot your target system and with the mount command, check that `/www/upload/files` is now a mount point for the third SD card partition. Also make sure that you can still upload new images, and that these images are listed in the web interface.

Adding a tmpfs partition for log files

For the moment, the upload script was storing its log file in `/www/upload/files/upload.log`. To avoid seeing this log file in the directory containing uploaded files, let's store it in `/var/log` instead.

Add the `/var/log/` directory to your root filesystem and modify the startup scripts to mount a `tmpfs` filesystem on this directory. You can test your `tmpfs` mount command line on the system before adding it to the startup script, in order to be sure that it works properly.

Modify the `www/cgi-bin/upload.cfg` configuration file to store the log file in `/var/log/upload.log`. You will lose your log file each time you reboot your system, but that's OK in our system. That's what `tmpfs` is for: temporary data that you don't need to keep across system reboots.

Reboot your system and check that it works as expected.

Making a SquashFS image

We are going to store the root filesystem in a SquashFS filesystem in the second partition of the SD card.

In order to create SquashFS images on your host, you need to install the `squashfs-tools` package. Now create a SquashFS image of your NFS root directory.

Finally, using the `dd` command, copy the file system image to the second partition of the SD card.

Booting on the SquashFS partition

In the U-boot shell, configure the kernel command line to use the second partition of the SD card as the root file system. Also add the `rootwait` boot argument, to wait for the SD card to be properly initialized before trying to mount the root filesystem. Since the SD cards are detected asynchronously by the kernel, the kernel might try to mount the root filesystem too early without `rootwait`.

Check that your system still works. Congratulations if it does!

Store the kernel image and DTB on the SD card

You'll first need to format the first partition, using:

```
sudo mkfs.vfat -F 16 -n boot /dev/mmcblk0p1
```

It will create a new FAT16 partition, called `boot`. Remove and plug the SD card back in. You can now copy the kernel image and Device Tree to it.

You now need to adjust the `bootcmd` of U-Boot so that it loads the kernel and DTB from the SD card instead of loading them from the NAND.

In U-boot, you can load a file from a FAT filesystem using a command like

```
fatload mmc 0:1 0x21000000 filename
```

Which will load the file named `filename` from the first partition of the device handled by the first MMC controller to the system memory at the address `0x21000000`.

Going further

At this point our board still uses the bootloaders (`at91bootstrap` and `U-Boot`) stored in the NAND flash. Let's try to have everything on our SD card. The ROM code can load the first stage bootloader from an MMC or SD card, from a file named `boot.bin` located in the first FAT partition. U-Boot will be stored as `u-boot.bin`.

For this you will need to recompile `at91bootstrap` (you'll need to switch to version 3.8.12) to support booting from an SD card. Then recompile U-Boot after reconfiguring it with its MMC configuration (we previously used the configuration for running from NAND flash).

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

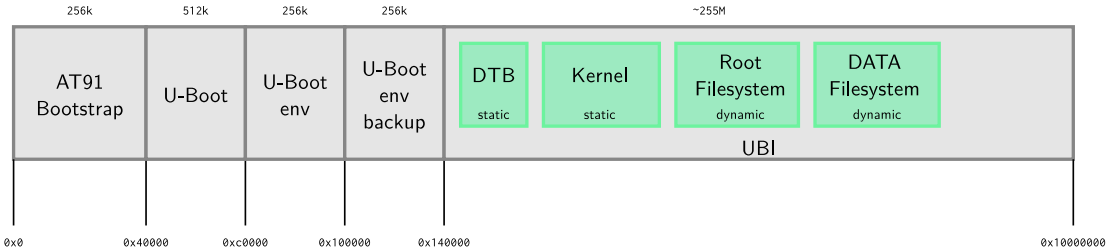
Instead of using an external SD card as in the previous lab, we will make our system use its internal flash storage.

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:



dynamic

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 won't use what you stored on the SD card during the previous lab. We will get back to the filesystem sources that you have 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. To find out this information, simply run `nand info` in U-Boot:

- The `Erase size` is 128KB, which is the size of the *Physical Erase Block*
- Both the `Page size` and `subpagesize` are 2KB, which means this NAND doesn't support sub-pages.

Therefore, the size of one *LEB* is the size of the *PEB* minus the size of two pages: 128 KB - 2 * 2 KB, i.e **124 KB**.

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` [15] on `www/upload/files`.

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

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.

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.

MTD partitioning and flashing

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. Once done, execute the `mtdparts` command and check the partition definitions in the output of this command.

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.

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

When flashing the UBI image, use the `trimffs` version of the command `nand write`[16].

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

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

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:

[16] 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.

- 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)

Third party libraries and applications

Objective: Learn how to leverage existing libraries and applications: how to configure, compile and install them

To illustrate how to use existing libraries and applications, we will extend the small root filesystem built in the *A tiny embedded system* lab to add the *ALSA* libraries and tools and an audio playback application using these libraries.

We'll see that manually re-using existing libraries is quite tedious, so that more automated procedures are necessary to make it easier. However, learning how to perform these operations manually will significantly help you when you face issues with more automated tools.

Audio support in the Kernel

Recompile your kernel with audio support. The options we want are: `CONFIG_SOUND`, `CONFIG_SND`, `CONFIG_SND_USB` and `CONFIG_SND_USB_AUDIO`.

At this stage, the easiest solution to update your kernel is probably to get back to copying it to RAM from `tftp`. Anyway, we will have to modify U-Boot environment variables, as we are going to switch back to NFS booting anyway.

Make sure that your board still boots with this new kernel.

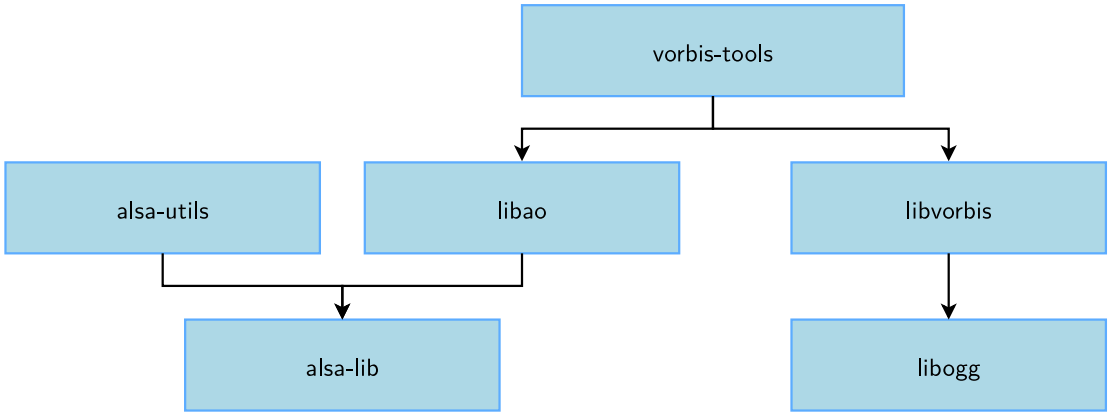
Figuring out library dependencies

We're going to integrate the `alsa-utils` and `ogg123` executables. As most software components, they in turn depend on other libraries, and these dependencies are different depending on the configuration chosen for them. In our case, the dependency chain for `alsa-utils` is quite simple, it only depends on the `alsa-lib` library.

The dependencies are a bit more complex for `ogg123`. It is part of `vorbis-tools`, that depend on `libao` and `libvorbis`. `libao` in turn depends on `alsa-lib`, and `libvorbis` on `libogg`.

`libao`, `alsa-utils` and `alsa-lib` are here to abstract the use of `ALSA`, one of the Audio Subsystems supported in Linux. `vorbis-tools`, `libvorbis` and `libogg` are used to handle the audio files encoded using the Ogg codec, which is quite common.

So, we end up with the following dependency tree:



Of course, all these libraries rely on the C library, which is not mentioned here, because it is already part of the root filesystem built in the *A tiny embedded system* lab. You might wonder how to figure out this dependency tree by yourself. Basically, there are several ways, that can be combined:

- Read the library documentation, which often mentions the dependencies;
- Read the help message of the `configure script` (by running `./configure --help`).
- By running the `configure` script, compiling and looking at the errors.

To configure, compile and install all the components of our system, we're going to start from the bottom of the tree with *alsa-lib*, then continue with *alsa-utils*, *libao*, *libogg*, and *libvorbis*, to finally compile *vorbis-tools*.

Preparation

For our cross-compilation work, we will need two separate spaces:

- A *staging* space in which we will directly install all the packages: non-stripped versions of the libraries, headers, documentation and other files needed for the compilation. This *staging* space can be quite big, but will not be used on our target, only for compiling libraries or applications;
- A *target* space, in which we will only copy the required files from the *staging* space: binaries and libraries, after stripping, configuration files needed at runtime, etc. This target space will take a lot less space than the *staging* space, and it will contain only the files that are really needed to make the system work on the target.

To sum up, the *staging* space will contain everything that's needed for compilation, while the *target* space will contain only what's needed for execution.

So, in `$HOME/embedded-linux-labs/thirdparty`, create two directories: `staging` and `target`.

For the target, we need a basic system with BusyBox and initialization scripts. We will re-use the system built in the *A tiny embedded system* lab, so copy this system in the target directory:

```
cp -a $HOME/embedded-linux-labs/tinysystem/nfsroot/* target/
```

Note that for this lab, a lot of typing will be required. To save time typing, we advise you to copy and paste commands from the electronic version of these instructions.

Testing

Make sure the `target/` directory is exported by your NFS server to your board by modifying `/etc/exports` and restarting your NFS server.

Make your board boot from this new directory through NFS.

alsa-lib

`alsa-lib` is a library supposed to handle the interaction with the ALSA subsystem. It is available at http://alsa-project.org. Download version 1.1.6, and extract it in `$HOME/embedded-linux-labs/thirdparty/`.

By looking at the `configure` script, we see that it has been generated by `autoconf` (the header contains a sentence like *Generated by GNU Autoconf 2.69*). Most of the time, `autoconf` comes with `automake`, that generates Makefiles from `Makefile.am` files. So alsa-lib uses a rather common build system. Let's try to configure and build it:

```
./configure
make
```

You can see that the files are getting compiled with gcc, which generates code for x86 and not for the target platform. This is obviously not what we want, so we clean-up the object and tell the configure script to use the ARM cross-compiler:

```
make clean
CC=arm-linux-gcc ./configure
```

Of course, the `arm-linux-gcc` cross-compiler must be in your `PATH` prior to running the configure script. The `CC` environment variable is the classical name for specifying the compiler to use.

Quickly, you should get an error saying:

```
checking whether we are cross compiling... configure: error: in `.../thirdparty/alsa-lib-1.1.6':
configure: error: cannot run C compiled programs.
If you meant to cross compile, use `--host'.
See `config.log' for more details
```

If you look at the `config.log` file, you can see that the `configure` script compiles a binary with the cross-compiler and then tries to run it on the development workstation. This is a rather usual thing to do for a `configure` script, and that's why it tests so early that it's actually doable, and bails out if not.

Obviously, it cannot work in our case, and the scripts exits. The job of the `configure` script is to test the configuration of the system. To do so, it tries to compile and run a few sample applications to test if this library is available, if this compiler option is supported, etc. But in our case, running the test examples is definitely not possible.

We need to tell the `configure` script that we are cross-compiling, and this can be done using the `--build` and `--host` options, as described in the help of the `configure` script:

```
System types:
  --build=BUILD configure for building on BUILD [guessed]
  --host=HOST cross-compile to build programs to run on HOST [BUILD]
```

The `--build` option allows to specify on which system the package is built, while the `--host` option allows to specify on which system the package will run. By default, the value of the `--build` option is guessed and the value of `--host` is the same as the value of the `--build` option. The value is guessed using the `./config.guess` script, which on your system should

return `i686-pc-linux-gnu`. See http://www.gnu.org/software/autoconf/manual/html_node/Specifying-Names.html for more details on these options.

So, let's override the value of the `--host` option:

```
CC=arm-linux-gcc ./configure --host=arm-linux
```

The `configure` script should end properly now, and create a Makefile. Run the `make` command, which should run just fine.

Look at the result of compiling in `src/.libs`: a set of object files and a set of `libasound.so*` files.

The `libasound.so*` files are a dynamic version of the library. The shared library itself is `libasound.so.2.0.0`, it has been generated by the following command line:

```
arm-linux-gcc -shared conf.o confmisc.o input.o output.o \
              async.o error.o dlmisc.o socket.o shmarea.o \
              userfile.o names.o -lm -ldl -lpthread -lrt  \
              -Wl,-soname -Wl,libasound.so.2 -o libasound.so.2.0.0
```

And creates the symbolic links `libasound.so` and `libasound.so.2`.

```
ln -s libasound.so.2.0.0 libasound.so.2
ln -s libasound.so.2.0.0 libasound.so
```

These symlinks are needed for two different reasons:

- `libasound.so` is used at compile time when you want to compile an application that is dynamically linked against the library. To do so, you pass the `-lLIBNAME` option to the compiler, which will look for a file named `lib<LIBNAME>.so`. In our case, the compilation option is `-lasound` and the name of the library file is `libasound.so`. So, the `libasound.so` symlink is needed at compile time;
- `libasound.so.2` is needed because it is the *SONAME* of the library. *SONAME* stands for *Shared Object Name*. It is the name of the library as it will be stored in applications linked against this library. It means that at runtime, the dynamic loader will look for exactly this name when looking for the shared library. So this symbolic link is needed at runtime.

To know what's the *SONAME* of a library, you can use:

```
arm-linux-readelf -d libasound.so.2.0.0
```

and look at the `(SONAME)` line. You'll also see that this library needs the C library, because of the `(NEEDED)` line on `libc.so.0`.

The mechanism of `SONAME` allows to change the library without recompiling the applications linked with this library. Let's say that a security problem is found in the alsa-lib release that provides libasound 2.0.0, and fixed in the next alsa-lib release, which will now provide libasound 2.0.1.

You can just recompile the library, install it on your target system, change the `libasound.so.2` link so that it points to `libasound.so.2.0.1` and restart your applications. And it will work, because your applications don't look specifically for `libasound.so.2.0.0` but for the *SONAME* `libasound.so.2`.

However, it also means that as a library developer, if you break the ABI of the library, you must change the *SONAME*: change from `libasound.so.2` to `libasound.so.3`.

Finally, the last step is to tell the `configure` script where the library is going to be installed. Most `configure` scripts consider that the installation prefix is `/usr/local/` (so that the library

is installed in `/usr/local/lib`, the headers in `/usr/local/include`, etc.). But in our system, we simply want the libraries to be installed in the `/usr` prefix, so let's tell the `configure` script about this:

```
CC=arm-linux-gcc ./configure --host=arm-linux --disable-python --prefix=/usr
make
```

For this library, this option may not change anything to the resulting binaries, but for safety, it is always recommended to make sure that the prefix matches where your library will be running on the target system.

Do not confuse the *prefix* (where the application or library will be running on the target system) from the location where the application or library will be installed on your host while building the root filesystem.

For example, libasound will be installed in `$HOME/embedded-linux-labs/thirdparty/target/usr/lib/` because this is the directory where we are building the root filesystem, but once our target system will be running, it will see libasound in `/usr/lib`.

The prefix corresponds to the path in the target system and **never** on the host. So, one should **never** pass a prefix like `$HOME/embedded-linux-labs/thirdparty/target/usr`, otherwise at runtime, the application or library may look for files inside this directory on the target system, which obviously doesn't exist! By default, most build systems will install the application or library in the given prefix (`/usr` or `/usr/local`), but with most build systems (including *autotools*), the installation prefix can be overridden, and be different from the configuration prefix.

We now only have the installation process left to do.

First, let's make the installation in the *staging* space:

```
make DESTDIR=$HOME/embedded-linux-labs/thirdparty/staging install
```

Now look at what has been installed by alsa-lib:

- Some configuration files in `/usr/share/alsa`
- The headers in `/usr/include`
- The shared library and its libtool (`.la`) file in `/usr/lib`
- A pkgconfig file in `/usr/lib/pkgconfig`. We'll come back to these later

Finally, let's install the library in the *target* space:

1. Create the `target/usr/lib` directory, it will contain the stripped version of the library
2. Copy the dynamic version of the library. Only `libasound.so.2` and `libasound.so.2.0.0` are needed, since `libasound.so.2` is the *SONAME* of the library and `libasound.so.2.0.0` is the real binary:
   - `cp -a staging/usr/lib/libasound.so.2* target/usr/lib`
3. Strip the library:
   - `arm-linux-strip target/usr/lib/libasound.so.2.0.0`

And we're done with alsa-lib!

Alsa-utils

Download alsa-utils from the ALSA offical webpage. We tested the lab with version 1.1.6.

Once uncompressed, we quickly discover that the alsa-utils build system is based on the *autotools*, so we will work once again with a regular `configure` script.

As we've seen previously, we will have to provide the prefix and host options and the CC variable:

```
CC=arm-linux-gcc ./configure --host=arm-linux --prefix=/usr
```

Now, we should quiclky get an error in the execution of the `configure` script:

```
checking for libasound headers version >= 1.0.27... not present.
configure: error: Sufficiently new version of libasound not found.
```

Again, we can check in `config.log` what the `configure` script is trying to do:

```
configure:7146: checking for libasound headers version >= 1.0.27
configure:7208: arm-linux-gcc -c -g -O2  conftest.c >&5
conftest.c:12:10: fatal error: alsa/asoundlib.h: No such file or directory
```

Of course, since *alsa-utils* uses *alsa-lib*, it includes its header file! So we need to tell the C compiler where the headers can be found: there are not in the default directory `/usr/include/`, but in the `/usr/include` directory of our *staging* space. The help text of the `configure` script says:

```
  CPPFLAGS            C/C++/Objective C preprocessor flags,
                      e.g. -I<include dir> if you have headers
                      in a nonstandard directory <include dir>
```

Let's use it:

```
CPPFLAGS=-I$HOME/embedded-linux-labs/thirdparty/staging/usr/include \
CC=arm-linux-gcc \
./configure --host=arm-linux --prefix=/usr
```

Now, it should stop a bit later, this time with the error:

```
checking for libasound headers version >= 1.0.27... found.
checking for snd_ctl_open in -lasound... no
configure: error: No linkable libasound was found.
```

The `configure` script tries to compile an application against *libasound* (as can be seen from the `-lasound` option): *alsa-utils* uses *alsa-lib*, so the `configure` script wants to make sure this library is already installed. Unfortunately, the `ld` linker doesn't find it. So, let's tell the linker where to look for libraries using the `-L` option followed by the directory where our libraries are (in `staging/usr/lib`). This `-L` option can be passed to the linker by using the `LDFLAGS` at configure time, as told by the help text of the `configure` script:

```
  LDFLAGS       linker flags, e.g. -L<lib dir> if you have
                libraries in a nonstandard directory <lib dir>
```

Let's use this `LDFLAGS` variable:

```
LDFLAGS=-L$HOME/embedded-linux-labs/thirdparty/staging/usr/lib \
CPPFLAGS=-I$HOME/embedded-linux-labs/thirdparty/staging/usr/include \
CC=arm-linux-gcc \
./configure --host=arm-linux --prefix=/usr
```


Once again, it should fail a bit further down the tests, this time complaining about a missing *curses helper header*. *curses* or *ncurses* is a graphical framework to design UIs in the terminal. This is only used by *alsamixer*, one of the tools provided by *alsa-utils*, that we are not going to use. Hence, we can just disable the build of *alsamixer*.

Of course, if we wanted it, we would have had to build *ncurses* first, just like we built *alsa-lib*. We will also need to disable support for *xmlto* that generates the documentation.

```
LDFLAGS=-L$HOME/embedded-linux-labs/thirdparty/staging/usr/lib \
CPPFLAGS=-I$HOME/embedded-linux-labs/thirdparty/staging/usr/include \
CC=arm-linux-gcc \
./configure --host=arm-linux --prefix=/usr \
--disable-alsamixer --disable-xmlto
```

Then, run the compilation with `make`. Hopefully, it works!

Let's now begin the installation process. Before really installing in the staging directory, let's install in a dummy directory, to see what's going to be installed (this dummy directory will not be used afterwards, it is only to verify what will be installed before polluting the staging space):

```
make DESTDIR=/tmp/alsa-utils/ install
```

The `DESTDIR` variable can be used with all Makefiles based on `automake`. It allows to override the installation directory: instead of being installed in the configuration prefix directory, the files will be installed in `DESTDIR/configuration-prefix`.

Now, let's see what has been installed in `/tmp/alsa-utils/` (run `tree /tmp/alsa-utils`):

```
./lib/udev/rules.d/90-alsa-restore.rules
./usr/bin/aseqnet
./usr/bin/aseqdump
./usr/bin/arecordmidi
./usr/bin/aplaymidi
./usr/bin/aconnect
./usr/bin/alsaloop
./usr/bin/speaker-test
./usr/bin/iecset
./usr/bin/aplay
./usr/bin/amidi
./usr/bin/amixer
./usr/bin/alsaucm
./usr/sbin/alsaconf
./usr/sbin/alsactl
./usr/share/sounds/alsa/Side_Left.wav
./usr/share/sounds/alsa/Rear_Left.wav
./usr/share/sounds/alsa/Noise.wav
./usr/share/sounds/alsa/Front_Right.wav
./usr/share/sounds/alsa/Front_Center.wav
./usr/share/sounds/alsa/Side_Right.wav
./usr/share/sounds/alsa/Rear_Right.wav
./usr/share/sounds/alsa/Rear_Center.wav
./usr/share/sounds/alsa/Front_Left.wav
./usr/share/locale/ru/LC_MESSAGES/alsaconf.mo
./usr/share/locale/ja/LC_MESSAGES/alsaconf.mo
./usr/share/locale/ja/LC_MESSAGES/alsa-utils.mo
./usr/share/locale/fr/LC_MESSAGES/alsa-utils.mo
```



```
./usr/share/locale/de/LC_MESSAGES/alsa-utils.mo
./usr/share/man/fr/man8/alsaconf.8
./usr/share/man/man8/alsaconf.8
./usr/share/man/man1/aseqnet.1
./usr/share/man/man1/aseqdump.1
./usr/share/man/man1/arecordmidi.1
./usr/share/man/man1/aplaymidi.1
./usr/share/man/man1/aconnect.1
./usr/share/man/man1/alsaloop.1
./usr/share/man/man1/speaker-test.1
./usr/share/man/man1/iecset.1
./usr/share/man/man1/aplay.1
./usr/share/man/man1/amidi.1
./usr/share/man/man1/amixer.1
./usr/share/man/man1/alsactl.1
./usr/share/alsa/speaker-test/sample_map.csv
./usr/share/alsa/init/ca0106
./usr/share/alsa/init/hda
./usr/share/alsa/init/test
./usr/share/alsa/init/info
./usr/share/alsa/init/help
./usr/share/alsa/init/default
./usr/share/alsa/init/00main
```

So, we have:

- The udev rules in `lib/udev`
- The alsa-utils binaries in `/usr/bin` and `/usr/sbin`
- Some sound samples in `/usr/share/sounds`
- The various translations in `/usr/share/locale`
- The manual pages in `/usr/share/man/`, explaining how to use the various tools
- Some configuration samples in `/usr/share/alsa`.

Now, let's make the installation in the *staging* space:

```
make DESTDIR=$HOME/embedded-linux-labs/thirdparty/staging/ install
```

Then, let's install only the necessary files in the *target* space, manually:

```
cd ..
cp -a staging/usr/bin/a* staging/usr/bin/speaker-test target/usr/bin/
cp -a staging/usr/sbin/alsa* target/usr/sbin
arm-linux-strip target/usr/bin/a*
arm-linux-strip target/usr/bin/speaker-test
arm-linux-strip target/usr/sbin/alsactl
mkdir -p target/usr/share/alsa/pcm
cp -a staging/usr/share/alsa/alsa.conf* target/usr/share/alsa/
cp -a staging/usr/share/alsa/cards target/usr/share/alsa/
cp -a staging/usr/share/alsa/pcm/default.conf target/usr/share/alsa/pcm/
```

And we're finally done with *alsa-utils*!

Now test that all is working fine by running the `speaker-test` util on your board, with the headset provided by your instructor plugged in. You may need to add the missing libraries from the toolchain install directory.

Caution: don't copy the `dmix.conf` file. `speaker-test` will tell you that it cannot find this file, but it won't work if you copy this file from the staging area.

The sound you get will be mainly noise (as what you would get by running `speaker-test` on your PCs). At least, sound output is showing some signs of life! It will get much better when we play samples with `ogg123`.

libogg

Now, let's work on *libogg*. Download the 1.3.3 version from http://xiph.org and extract it.

Configuring *libogg* is very similar to the configuration of the previous libraries:

```
CC=arm-linux-gcc ./configure --host=arm-linux --prefix=/usr
```

Of course, compile the library:

```
make
```

Installation to the *staging* space can be done using the classical `DESTDIR` mechanism:

```
make DESTDIR=$HOME/embedded-linux-labs/thirdparty/staging/ install
```

And finally, only install manually in the *target* space the files needed at runtime:

```
cd ..
cp -a staging/usr/lib/libogg.so.0* target/usr/lib/
arm-linux-strip target/usr/lib/libogg.so.0.8.3
```

Done with *libogg*!

libvorbis

Libvorbis is the next step. Grab the 1.3.6 version from http://xiph.org and uncompress it.

Once again, the *libvorbis* build system is a nice example of what can be done with a good usage of the autotools. Cross-compiling *libvorbis* is very easy, and almost identical to what we've seen with *alsa-utils*. First, the `configure` step:

```
CC=arm-linux-gcc \
./configure --host=arm-linux --prefix=/usr
```

It will fail with:

```
configure: error: Ogg >= 1.0 required !
```

By running `./configure --help`, you will find the `--with-ogg-libraries` and `--with-ogg-includes` options. Use these:

```
CC=arm-linux-gcc ./configure --host=arm-linux --prefix=/usr \
  --with-ogg-includes=$HOME/embedded-linux-labs/thirdparty/staging/usr/include \
  --with-ogg-libraries=$HOME/embedded-linux-labs/thirdparty/staging/usr/lib
```

Then, compile the library:

```
make
```


Install it in the *staging* space:

```
make DESTDIR=$HOME/embedded-linux-labs/thirdparty/staging/ install
```

And install only the required files in the *target* space:

```
cd ..
cp -a staging/usr/lib/libvorbis.so.0* target/usr/lib/
arm-linux-strip target/usr/lib/libvorbis.so.0.4.8
cp -a staging/usr/lib/libvorbisfile.so.3* target/usr/lib/
arm-linux-strip target/usr/lib/libvorbisfile.so.3.3.7
```

And we're done with *libvorbis*!

libao

Now, let's work on *libao*. Download the 1.2.0 version from http://xiph.org and extract it.

Configuring *libao* is once again fairly easy, and similar to every sane autotools based build system:

```
LDFLAGS=-L$HOME/embedded-linux-labs/thirdparty/staging/usr/lib \
CPPFLAGS=-I$HOME/embedded-linux-labs/thirdparty/staging/usr/include \
CC=arm-linux-gcc ./configure --host=arm-linux \
                             --prefix=/usr
```

Of course, compile the library:

```
make
```

Installation to the *staging* space can be done using the classical `DESTDIR` mechanism:

```
make DESTDIR=$HOME/embedded-linux-labs/thirdparty/staging/ install
```

And finally, install manually the only needed files at runtime in the *target* space:

```
cd ..
cp -a staging/usr/lib/libao.so.4* target/usr/lib/
arm-linux-strip target/usr/lib/libao.so.4.1.0
```

We will also need the alsa plugin that is loaded dynamically by *libao* at startup:

```
mkdir -p target/usr/lib/ao/plugins-4/
cp -a staging/usr/lib/ao/plugins-4/libalsa.so target/usr/lib/ao/plugins-4/
```

Done with *libao*!

vorbis-tools

Finally, thanks to all the libraries we compiled previously, all the dependencies are ready. We can now build the vorbis tools themselves. Download the 1.4.0 version from the official website, at http://xiph.org/. As usual, extract the tarball.

Before starting the configuration, let's have a look at the available options by running `./configure --help`. Many options are available. We see that we can, in addition to the usual autotools configuration options:

- Enable/Disable the various tools that are going to be built: `ogg123`, `oggdec`, `oggenc`, etc.
- Enable or disable support for various other codecs: FLAC, Speex, etc.

- Enable or disable the use of various libraries that can optionally be used by the vorbis tools

So, let's begin with our usual `configure` line:

```
LDFLAGS=-L$HOME/embedded-linux-labs/thirdparty/staging/usr/lib \
CPPFLAGS=-I$HOME/embedded-linux-labs/thirdparty/staging/usr/include \
CC=arm-linux-gcc \
./configure --host=arm-linux --prefix=/usr
```

At the end, you should see the following warning:

```
configure: WARNING: Prerequisites for ogg123 not met, ogg123 will be skipped.
Please ensure that you have POSIX threads, libao, and (optionally) libcurl
libraries and headers present if you would like to build ogg123.
```

Which is unfortunate, since we precisely want `ogg123`.

If you look back at the script output, you should see that at some point that it tests for *libao* and fails to find it:

```
checking for AO... no
configure: WARNING: libao too old; >= 1.0.0 required
```

If you look into the `config.log` file now, you should find something like:

```
configure:22343: checking for AO
configure:22351: $PKG_CONFIG --exists --print-errors "ao >= 1.0.0"
Package ao was not found in the pkg-config search path.
Perhaps you should add the directory containing `ao.pc'
to the PKG_CONFIG_PATH environment variable
No package 'ao' found
```

In this case, the `configure` script uses the *pkg-config* system to get the configuration parameters to link the library against *libao*. By default, *pkg-config* looks in `/usr/lib/pkgconfig/` for `.pc` files, and because the `libao-dev` package is probably not installed in your system the `configure` script will not find *libao* library that we just compiled.

It would have been worse if we had the package installed, because it would have detected and used our host package to compile *libao*, which, since we're cross-compiling, is a pretty bad thing to do.

This is one of the biggest issue with cross-compilation: mixing host and target libraries, because build systems have a tendency to look for libraries in the default paths.

So, now, we must tell *pkg-config* to look inside the `/usr/lib/pkgconfig/` directory of our *staging* space. This is done through the `PKG_CONFIG_LIBDIR` environment variable, as explained in the manual page of `pkg-config`.

Moreover, the `.pc` files contain references to paths. For example, in `$HOME/embedded-linux-labs/thirdparty/staging/usr/lib/pkgconfig/ao.pc`, we can see:

```
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include
[...]
Libs: -L${libdir} -lao
Cflags: -I${includedir}
```


So we must tell `pkg-config` that these paths are not absolute, but relative to our *staging* space. This can be done using the `PKG_CONFIG_SYSROOT_DIR` environment variable.

Then, let's run the configuration of the vorbis-tools again, passing the `PKG_CONFIG_LIBDIR` and `PKG_CONFIG_SYSROOT_DIR` environment variables:

```
LDFLAGS=-L$HOME/embedded-linux-labs/thirdparty/staging/usr/lib \
CPPFLAGS=-I$HOME/embedded-linux-labs/thirdparty/staging/usr/include \
PKG_CONFIG_LIBDIR=$HOME/embedded-linux-labs/thirdparty/staging/usr/lib/pkgconfig \
PKG_CONFIG_SYSROOT_DIR=$HOME/embedded-linux-labs/thirdparty/staging \
CC=arm-linux-gcc \
./configure --host=arm-linux --prefix=/usr
```

Now, the `configure` script should end properly, we can now start the compilation:

```
make
```

It should fail with the following cryptic message:

```
make[2]: Entering directory '/home/tux/embedded-linux-labs/thirdparty/vorbis-tools-1.4.0/ogg123'
if arm-linux-gcc -DSYSCONFDIR=\"/usr/etc\" -DLOCALEDIR=\"/usr/share/locale\" -DHAVE_CONFIG_H -I. -I. -I.. -I/usr/include -I..
then mv -f ".deps/audio.Tpo" ".deps/audio.Po"; else rm -f ".deps/audio.Tpo"; exit 1; fi
In file included from audio.c:22:
/usr/include/stdio.h:27:10: fatal error: bits/libc-header-start.h: No such file or directory
```

You can notice that `/usr/include` is added to the include paths. Again, this is not what we want because it contains includes for the host, not the target. It is coming from the autodetected value for `CURL_CFLAGS`.

Add the `--without-curl` option to the `configure` invocation, restart the compilation.

The compilation may then fail with an error related to *libm*. While the code uses the function from this library, the generated Makefile doesn't give the right command line argument in order to link against the libm.

If you look at the `configure` help, you can see

```
LIBS        libraries to pass to the linker, e.g. -l<library>
```

And this is exactly what we are supposed to use to add new linker flags.

Add this to the `configure` command line to get

```
LDFLAGS=-L$HOME/embedded-linux-labs/thirdparty/staging/usr/lib \
CPPFLAGS=-I$HOME/embedded-linux-labs/thirdparty/staging/usr/include \
PKG_CONFIG_LIBDIR=$HOME/embedded-linux-labs/thirdparty/staging/usr/lib/pkgconfig \
PKG_CONFIG_SYSROOT_DIR=$HOME/embedded-linux-labs/thirdparty/staging \
LIBS=-lm \
CC=arm-linux-gcc \
./configure --host=arm-linux --prefix=/usr --without-curl
```

Finally, it builds!

Now, install the vorbis-tools to the *staging* space using:

```
make DESTDIR=$HOME/embedded-linux-labs/thirdparty/staging/ install
```

And then install them in the *target* space:

```
cd ..
cp -a staging/usr/bin/ogg* target/usr/bin
arm-linux-strip target/usr/bin/ogg*
```


You can now test that everything works! Run `ogg123` on the sample file found in `thirdparty/data`.

There should still be one missing C library object. Copy it, and you should get: +

```
ERROR: Failed to load plugin /usr/lib/ao/plugins-4/libalsa.so => dlopen() failed
=== Could not load default driver and no driver specified in config file. Exiting.
```

This error message is unfortunately not sufficient to figure out what's going wrong. It's a good opportunity to use the `strace` utility to get more details about what's going on. To do so, you can used the one built by Crosstool-ng inside the toolchain `target/usr/bin` directory.

You can now run `ogg123` through `strace`:

```
strace ogg123 /sample.ogg
```

You can see that the command fails to open the `ld-uClibc.so.1` file:

```
open("/lib/ld-uClibc.so.1", O_RDONLY)   = -1 ENOENT (No such file or directory)
open("/lib/ld-uClibc.so.1", O_RDONLY)   = -1 ENOENT (No such file or directory)
open("/usr/lib/ld-uClibc.so.1", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/X11R6/lib/ld-uClibc.so.1", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/home/tux/embedded-linux-labs/thirdparty/staging/usr/lib/ld-uClibc.so.1", O_RDONLY) = -1 ENOENT (No
write(2, "ERROR: Failed to load plugin ", 29ERROR: Failed to load plugin ) = 29
write(2, "/usr/lib/ao/plugins-4/libalsa.so", 32/usr/lib/ao/plugins-4/libalsa.so) = 32
write(2, " => dlopen() failed\n", 20 => dlopen() failed
```

Now, look for `ld-uClibc.so.1` in the toolchain. You can see that both `ld-uClibc.so.1` and `ld-uClibc.so.0` are symbolic links to the same file. So, create the missing link under `target/lib` and run `ogg123` again.

Everything should work fine now. Enjoy the sound sample!

To finish this lab completely, and to be consistent with what we've done before, let's strip the libraries in `target/lib`:

```
arm-linux-strip target/lib/*
```


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 https://buildroot.org/. Download the latest stable 2019.02 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 already existing 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: Custom toolchain`
  - `Toolchain path`: use the toolchain you built: `/home/<user>/x-tools/arm-training-linux-uclibcgnueabihf` (replace `<user>` by your actual user name)
  - `External toolchain gcc version: 8.x`
  - `External toolchain kernel headers series: 4.16.x`
  - `External toolchain C library: uClibc/uClibc-ng`
  - We must tell Buildroot about our toolchain configuration, so select `Toolchain has WCHAR support?`, `Toolchain has SSP support?` and `Toolchain has C++ support?`. Buildroot will check these parameters anyway.
- `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 `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-2019.02/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 `ogg123` like you used to in the previous lab.

Going further

- 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.

- 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.
- *Only for the 5 day course, covering flash storage*
  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, the minimum I/O unit size is 2048 and the Maximum logical eraseblock (LEB) count is 133.
  - Then, once the image has been generated, update your rootfs volume.

Application development

Objective: Compile and run your own ncurses application on the target.

Setup

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

Compile your own application

We will re-use the system built during the *Buildroot lab* and add to it our own application.

In the lab directory the file `app.c` contains a very simple *ncurses* application. It is a simple game where you need to reach a target using the arrow keys of your keyboard. We will compile and integrate this simple application to our Linux system.

Buildroot has generated toolchain wrappers in `output/host/usr/bin`, which make it easier to use the toolchain, since these wrappers pass some mandatory flags (especially the `--sysroot` *gcc* flag, which tells *gcc* where to look for the headers and libraries).

Let's add this directory to our `PATH`:

```
export PATH=$HOME/embedded-linux-labs/buildroot/buildroot-XXXX.YY/output/host/usr/bin:$PATH
```

Let's try to compile the application:

```
arm-linux-gcc -o app app.c
```

It complains about undefined references to some symbols. This is normal, since we didn't tell the compiler to link with the necessary libraries. So let's use `pkg-config` to query the *pkg-config* database about the location of the header files and the list of libraries needed to build an application against *ncurses*[17]:

```
arm-linux-gcc -o app app.c $(pkg-config --libs --cflags ncurses)
```

You can see that `ncurses` doesn't need anything in particular for the `CFLAGS` but you can have a look at what is needed for *libvorbis* to get a feel of what it can look like:

```
pkg-config --libs --cflags vorbis
```

Our application is now compiled! Copy the generated binary to the NFS root filesystem (in the `root/` directory for example), start your system, and run your application!

You can also try to run it over ssh if you added ssh support to your target. Do you notice the difference?

[17] Again, `output/host/usr/bin` has a special `pkg-config` that automatically knows where to look, so it already knows the right paths to find `.pc` files and their sysroot.

Remote application debugging

Objective: Use `strace` *to diagnose program issues. Use* `gdbserver` *and a cross-debugger to remotely debug an embedded application*

Setup

Go to the `$HOME/embedded-linux-labs/debugging` directory. Create an `nfsroot` directory.

Debugging setup

Because of issues in *gdb* and *ltrace* in the uClibc version that we are using in our toolchain, we will use a different toolchain in this lab, based on *glibc*.

As *glibc* has more complete features that lighter libraries, it looks like a good idea to do your application debugging work with a *glibc* toolchain first, and then switch to lighter libraries once your application and software stack is production ready.

Extract the Buildroot 2019.02 sources into the current directory.

Then, in the `menuconfig` interface, configure the target architecture as done previously but configure the toolchain and target packages differently:

- In `Toolchain`:
  - `Toolchain type`: `External toolchain`
  - `Toolchain`: `Custom Toolchain`
  - `Toolchain origin`: `Toolchain to be downloaded and installed`
  - `Toolchain URL`: `https://toolchains.bootlin.com/downloads/releases/toolchains/armv7-eabihf/tarballs/armv7-eabihf--glibc--bleeding-edge-2018.07-3.tar.bz2` You can easily choose such a toolchain on https://toolchains.bootlin.com by selecting the architecture, the C library and the compiler version you need. While you can try with other versions, the above toolchain is known to make this lab work.
  - `External toolchain gcc version`: `8.x`
  - `External toolchain kernel headers series`: `4.14.x`
  - `External toochain C library`: `glibc/eglibc`
  - Select `Toolchain has SSP support?`
  - Select `Toolchain has RPC support?`
  - Select `Toolchain has C++ support?`
  - Select `Copy gdb server to the Target`
- `Target packages`
  - `Debugging, profiling and benchmark`

* Select `ltrace`
* Select `strace`

Now, build your root filesystem.

Go back to the `$HOME/embedded-linux-labs/debugging` directory and extract the `buildroot-2019.02/output/images/rootfs.tar` archive in the `nfsroot` directory.

Add this directory to the `/etc/exports` file and restart `nfs-kernel-server`.

Boot your ARM board over NFS on this new filesystem, using the same kernel as before.

Using strace

Now, go to the `$HOME/embedded-linux-labs/debugging` directory.

`strace` allows to trace all the system calls made by a process: opening, reading and writing files, starting other processes, accessing time, etc. When something goes wrong in your application, strace is an invaluable tool to see what it actually does, even when you don't have the source code.

Update the PATH:

```
export PATH=$HOME/embedded-linux-labs/debugging/buildroot-2019.02/output/host/bin:$PATH
```

With your cross-compiling toolchain compile the `data/vista-emulator.c` program, strip it with `arm-linux-strip`, and copy the resulting binary to the `/root` directory of the root filesystem.

Back to target system, try to run the `/root/vista-emulator` program. It should hang indefinitely!

Interrupt this program by hitting `[Ctrl] [C]`.

Now, running this program again through the `strace` command and understand why it hangs. You can guess it without reading the source code!

Now add what the program was waiting for, and now see your program proceed to another bug, failing with a segmentation fault.

Using ltrace

Now run the program through `ltrace`.

Now you should see what the program does: it tries to consume as much system memory as it can!

Also run the program through `ltrace -c`, to see what function call statistics this utility can provide.

It's also interesting to run the program again with `strace`. You will see that memory allocations translate into `mmap()` system calls. That's how you can recognize them when you're using `strace`.

Using gdbserver

We are now going to use `gdbserver` to understand why the program segfaults.

Compile `vista-emulator.c` again with the `-g` option to include debugging symbols. This time, just keep it on your workstation, as you already have the version without debugging symbols on your target.

Then, on the target side, run `vista-emulator` under `gdbserver`. `gdbserver` will listen on a TCP port for a connection from `gdb`, and will control the execution of `vista-emulator` according to the `gdb` commands:

```
gdbserver localhost:2345 vista-emulator
```

On the host side, run `arm-linux-gdb` (also found in your toolchain):

```
arm-linux-gdb vista-emulator
```

You can also start the debugger through the `ddd` interface:

```
ddd --debugger arm-linux-gdb vista-emulator
```

`gdb` starts and loads the debugging information from the `vista-emulator` binary that has been compiled with `-g`.

Then, we need to tell where to find our libraries, since they are not present in the default `/lib` and `/usr/lib` directories on your workstation. This is done by setting the `gdb sysroot` variable (on one line):

```
(gdb) set sysroot /home/<user>/embedded-linux-labs/debugging/
buildroot-2019.02/output/staging
```

Of course, replace `<user>` by your actual user name.

And tell `gdb` to connect to the remote system:

```
(gdb) target remote <target-ip-address>:2345
```

Then, use `gdb` as usual to set breakpoints, look at the source code, run the application step by step, etc. Graphical versions of `gdb`, such as `ddd` can also be used in the same way. In our case, we'll just start the program and wait for it to hit the segmentation fault:

```
(gdb) continue
```

You could then ask for a backtrace to see where this happened:

```
(gdb) backtrace
```

This will tell you that the segmentation fault occurred in a function of the C library, called by our program. This should help you in finding the bug in our application.

What to remember

During this lab, we learned that...

- It's easy to study the behavior of programs and diagnose issues without even having the source code, thanks to strace.
- You can leave a small `gdbserver` program (about 300 KB) on your target that allows to debug target applications, using a standard `gdb` debugger on the development host.
- It is fine to strip applications and binaries on the target machine, as long as the programs and libraries with debugging symbols are available on the development host.

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

Create an `nfsroot` directory.

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.

Download and extract the Buildroot 2016.02 sources. As the latest version of Xenomai doesn't seem to work on the Xplained board (yet), we need an older version of Buildroot that will build Xenomai 2.6.

Configure Buildroot with the following settings, using the `/` command in `make menuconfig` to find parameters by their name:

- In `Target`:
  - `Target architecture: ARM (little endian)`
  - `Target Architecture Variant: cortex-A5`
- In `Toolchain`:
  - `Toolchain type: External toolchain`
  - `Toolchain: Sourcery CodeBench ARM 2014.05`
- 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`[18] are enabled.

Now, build your root filesystem.

As you are using a 64 bit distribution, Buildroot should also ask you to install 32 bit compatibility packages to be able to execute the Sourcery CodeBench external toolchain:

```
sudo apt install libc6-i386 lib32stdc++6 lib32z1
```

At the end of the build job, extract the `output/images/rootfs.tar` archive in the `nfsroot` directory.

The last thing to do is to add a few files that we will need in our tests:

```
cp data/* nfsroot/root
```

Downloading sources and patches

We will use a kernel version that is supported by the PREEMPT_RT patchset.

So, go to https://kernel.org/pub/linux/kernel/projects/rt/4.13/, download the latest patch available in a single file.

Then go to http://kernel.org and download the exact version corresponding to the patch you downloaded. At the time of this writing, this version was 4.13.10.

Compile a standard Linux kernel

Extract the sources of your 4.13.x kernel but don't apply the PREEMPT_RT patches yet.

Configure your kernel for your Xplained board, and then make sure that the below settings are disabled: `CONFIG_PROVE_LOCKING`, `CONFIG_DEBUG_LOCK_ALLOC`, `CONFIG_DEBUG_MUTEXES` and `CONFIG_DEBUG_SPINLOCK`.

Also, for the moment, disable the `CONFIG_HIGH_RES_TIMERS` option which impact we want to measure.

Boot the Xplained board by mounting the root filesystem that you built. As usual, login as `root`, there is no password.

[18] Needed by the Xenomai testsuite.

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=$HOME/embedded-linux-labs/realtime/buildroot-YYYY.MM/output/host/usr/bin:$PATH
```

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

Now compile this program:

```
arm-none-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`.

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 add back the `CONFIG_HIGH_RES_TIMERS` option in the kernel configuration.

Recompile your kernel, boot your Xplained 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.

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

Download the latest `PREEMPT_RT` kernel patch and apply it to your kernel sources.

Configure your kernel with `CONFIG_PREEMPT_RT_FULL` and boot it.

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

Testing Xenomai scheduling latency

Stay in `$HOME/embedded-linux-labs/realtime`.

Download the 2.6.4 release of Xenomai (that's what our version of Buildroot supports by default), and extract it.

As you can see in the `ksrc/arch/arm/patches` directory, the most recent Linux kernel version supported by Xenomai for ARM is 3.14.17.

Then, download the 3.14.17 Linux sources (**not the latest 3.14.x sources** because the Xenomai patches only apply to this exact version), and extract them.

Now, prepare our kernel for Xenomai compilation:

```
cd $HOME/embedded-linux-labs/realtime
./xenomai-2.6.4/scripts/prepare-kernel.sh --arch=arm \
--linux=linux-3.14.17 \
--adeos=xenomai-2.6.4/ksrc/arch/arm/patches/ipipe-core-3.14.17-arm-4.patch
```

Now, configure your kernel for SAMA5 boards, then start the kernel configuration interface, and make sure that the below options are enabled, taking your time to read their description:

- `CONFIG_XENOMAI`
- `CONFIG_XENO_DRIVERS_TIMERBENCH`
- `CONFIG_XENO_HW_UNLOCKED_SWITCH`

Compile your kernel, using the same Sourcery CodeBench compiler as earlier in the lab[19].

While the kernel compiles, we can start to build our application against the Xenomai libraries. We will need *pkg-config* built by Buildroot. So go in your Buildroot source directory, and force Buildroot to build the host variant of *pkg-config*:

```
cd $HOME/embedded-linux-labs/realtime/buildroot-YYYY.MM/
make host-pkgconf
```

We can now compile `rttest` for the Xenomai POSIX skin:

```
cd $HOME/embedded-linux-labs/realtime/nfsroot/root
export PATH=$HOME/embedded-linux-labs/realtime/buildroot-YYYY.MM/output/host/usr/bin:$PATH
arm-none-linux-gnueabi-gcc -o rttest rttest.c \
  $(pkg-config --libs --cflags libxenomai_posix)
```

Now boot the board with the new kernel.

Run the following commands on the board:

```
echo 0 > /proc/xenomai/latency
```

This will disable the timer compensation feature of Xenomai. This feature allows Xenomai to adjust the timer programming to take into account the time the system needs to schedule a task after being woken up by a timer. However, this feature needs to be calibrated specifically for each system. By disabling this feature, we will have raw Xenomai results, that could be further improved by doing proper calibration of this compensation mechanism.

[19] Your own toolchain is too recent for the 3.14 kernel, which doesn't support compiling with gcc5 yet. The Sourcery CodeBench gcc version is 4.8.x.

Run the tests again, compare the results.

Testing Xenomai interrupt latency

Measure the interrupt latency with and without load, running the following command:

```
latency -t 2
```