Embedded Linux Training

Lab Book

Free Electrons
http://free-electrons.com

March 29, 2014

About this document

Updates to this document can be found on http://free-electrons.com/doc/training/embedded-linux/.

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

More details about our training sessions can be found on http://free-electrons.com/training.

Copying this document

Training setup

Download files and directories used in practical labs

Install lab data

For the different labs in the training, 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 http://free-electrons.com/doc/training/embedded-linux/labs.tar.xz
sudo tar Jvxf labs.tar.xz
sudo chown -R <user>.<user> felabs
```

Note that using `root` permissions are required to extract the character and block device files contained in this lab archive. This is an exception. For all the other archives that you will handle during the practical labs, you will never need `root` permissions to extract them. If there is another exception, we will let you know.

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

You are now ready to start the real practical labs!

Install extra packages

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

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

More guidelines

Can be useful throughout any of the labs

- Read instructions and tips carefully. Lots of people make mistakes or waste time because they missed an explanation or a guideline.
- Always read error messages carefully, in particular the first one which is issued. Some people stumble on very simple errors just because they specified a wrong file path and didn't pay enough attention to the corresponding error message.
- Never stay stuck with a strange problem more than 5 minutes. Show your problem to your colleagues or to the instructor.
- You should only use the `root` user for operations that require super-user privileges, such as: mounting a file system, loading a kernel module, changing file ownership, configur-

ing the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.

- If you ran commands from a root shell by mistake, your regular user may no longer be able to handle the corresponding generated files. In this case, use the `chown -R` command to give the new files back to your regular user.
  Example: `chown -R myuser.myuser linux-3.4`

Building a cross-compiling toolchain

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

After this lab, you will be able to:

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

Setup

Go to the `$HOME/felabs/sysdev/toolchain` directory.

Install needed packages

Install the packages needed for this lab:

```
sudo apt-get install autoconf automake libtool libexpat1-dev \
     libncurses5-dev bison flex patch curl cvs texinfo \
     build-essential subversion gawk python-dev gperf
```

Getting Crosstool-ng

Get the latest 1.19.x release of Crosstool-ng at http://crosstool-ng.org. Expand the archive right in the current directory, and enter the Crosstool-ng source directory.

Installing Crosstool-ng

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

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

Then you can get Crosstool-ng help by running

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

Configure the toolchain to produce

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

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

We will use the arm-unknown-linux-uclibcgnueabi sample. It can be loaded by issuing:

```
./ct-ng arm-unknown-linux-uclibcgnueabi
```

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

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

In `Path and misc options`:

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

In `Toolchain options`:

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

In `Debug facilities`:

- Make sure that `gdb`, `strace` and `ltrace` are enabled.
- Remove the other options (`dmalloc` and `duma`).
- In `gdb` options:
  - Make sure that the `Cross-gdb` and `Build a static gdbserver` options are enabled; the other options are not needed.
  - Set `gdb version` to `7.4.1`.

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

Produce the toolchain

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

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

Nothing is simpler:

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

And wait!

Known issues

Source archives not found on the Internet

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

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

ppl-0.10.2 compiling error with gcc 4.7.1

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

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

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

Testing the toolchain

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

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

Cleaning up

To save about 3 GB of storage space, do a `./ct-ng clean` in the Crosstool-NG source directory. This will remove the source code of the different toolchain components, as well as all the generated files that are now useless since the toolchain has been installed in `/usr/local/xtools`.

Bootloader - U-Boot

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

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

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

The IGEPv2 board, which uses the DM3730 or the OMAP3530 processors, 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. Therefore, by using an MMC/SD card, we can start up a OMAP3-based board without having anything installed on it.

Setup

Go to the `~/felabs/sysdev/bootloader` directory.

MMC/SD card setup

The ROM monitor can read files from a FAT filesystem on the MMC/SD card. However, the MMC/SD card must be carefully partitionned, and the filesystem carefully created in order to be recognized by the ROM monitor. Here are special instructions to format an MMC/SD card for the OMAP-based platforms.

First, connect your card reader to your workstation, with the MMC/SD card inside. Type the `dmesg` command to see which device is used by your workstation. In case the device is `/dev/sdb`, you will see something like:

```
sd 3:0:0:0: [sdb] 3842048 512-byte hardware sectors: (1.96 GB/1.83 GiB)
```

If your PC has an internal MMC/SD card reader, the device may also been seen as `/dev/mmcblk0`, and the first partition as `mmcblk0p1`. [1]. You will see that the MMC/SD card is seen in the same way by the IGEPv2 board.

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

[1]This is not always the case with internal MMC/SD card readers. On some PCs, such devices are behind an internal USB bus, and thus are visible in the same way external card readers are

Caution: read this carefully before proceeding. You could destroy existing partitions on your PC!
Do not make the confusion between the device that is used by your board to represent your MMC/SD disk (probably `/dev/sda`), and the device that your workstation uses when the card reader is inserted (probably `/dev/sdb`).
So, don't use the `/dev/sda` device to reflash your MMC disk from your workstation. People have already destroyed their Windows partition by making this mistake.

You can also run `cat /proc/partitions` to list all block devices in your system. Again, make sure to distinguish the SD/MMC card from the hard drive of your development workstation!

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

```
$ sudo umount /dev/sdb1
$ sudo umount /dev/sdb2
...
```

Now, clear possible MMC/SD card contents remaining from previous training sessions:

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

As we explained earlier, the TI OMAP rom monitor needs special partition geometry settings to read partition contents. The MMC/SD card must have 255 heads and 63 sectors.

Let's use the `cfdisk` command to create a first partition with these settings:

```
sudo cfdisk -h 255 -s 63 /dev/sdb
```

In the `cfdisk` interface, create a first primary partition, starting from the beginning, with a 64 MB size, a `Bootable` type and a `0C` type (`W95 FAT32 (LBA)`). Press `Write` when you are done.

If you used `fdisk` before, you should find `cfdisk` much more convenient!

Format this new partition in FAT32, with the `boot` label (name):

```
sudo mkfs.vfat -n boot -F 32 /dev/sdb1
```

Then, remove and insert your card again.

Your MMC/SD card is ready to use.

U-Boot setup

Download U-Boot from the mainline igep download site:

```
wget ftp://ftp.denx.de/pub/u-boot/u-boot-2013.10.tar.bz2
tar xvf u-boot-2013.10.tar.bz2
cd u-boot-2013.10
```

Then, apply the `0001-arm-omap-i2c-don-t-zero-cnt-in-i2c_write.patch` patch from this lab's `data` directory:

```
cat /path/to/0001-arm-omap-i2c-don-t-zero-cnt-in-i2c_write.patch | \
   patch -p1
```

Get an understanding of its 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>_config`, where `<NAME>` is the name of your board as declared in the `boards.cfg` file. There are two flavors of the IGEPv2: since the RevC6 they use a NAND flash (`igep0020_nand`) and before this revision they were a OneNAND flash (`igep0020`). Note that for our platform, the configuration file is `include/configs/igep00x0.h`. Read this file to get an idea of how a U-Boot configuration file is written;
- Finally, run `make`[2], which should build U-Boot.

You can now copy the generated `MLO` and `u-boot.img` files to the MMC card. `MLO` is the first stage bootloader, `u-boot.img` is the second stage bootloader.

Unmount the MMC card partition.

Setting up serial communication with the board

Plug the IGEPv2 board on your computer using the provided USB-to-serial cable. When plugged-in, a serial port should appear, `/dev/ttyUSB0`.

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

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

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

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

```
sudo adduser $USER dialout
```

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

Run `picocom -b 115200 /dev/ttyUSB0`, to start serial communication on `/dev/ttyUSB0`, with a baudrate of 115200. If you wish to exit `picocom`, press `[Ctrl][a]` followed by `[Ctrl][x]`.

Testing U-Boot on the MMC card

Insert the MMC card into the IGEP board, reset the board and check that it boots your new bootloaders. You can verify this by checking the build dates:

```
U-Boot SPL 2013.10 (Nov 15 2013 - 14:12:51)
reading u-boot.img
reading u-boot.img


U-Boot 2013.10 (Nov 15 2013 - 14:12:51)

OMAP36XX/37XX-GP ES1.2, CPU-OPP2, L3-200MHz, Max CPU Clock 1 Ghz
IGEPv2 + LPDDR/NAND
I2C:   ready
DRAM:  512 MiB
```

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

```
NAND:  512 MiB
MMC:   OMAP SD/MMC: 0
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Die ID #415c00029ff80000015913d80702502a
Net:   smc911x-0
Hit any key to stop autoboot:  0
```

The message `reading u-boot.img` also confirms that U-Boot has been loaded from the MMC device. You don't get it when you boot from NAND flash (there are no files on raw flash anyway).

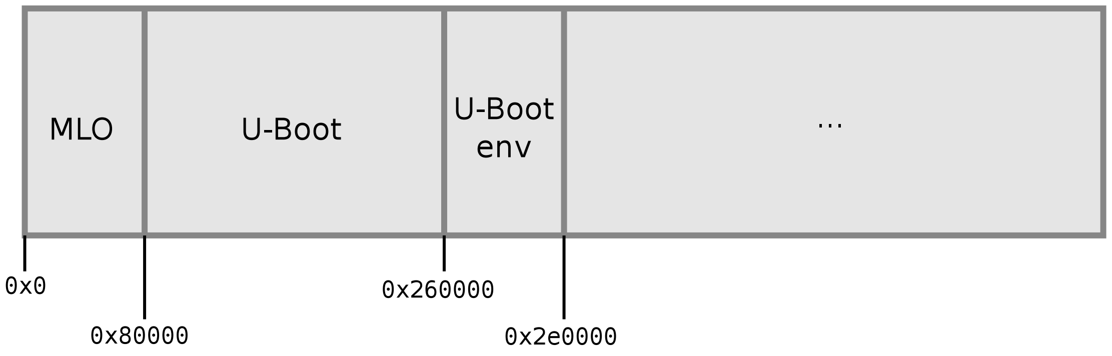
Interrupt the countdown to enter the U-Boot shell:

```
U-Boot #
```

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

Reflashing from U-boot

We will flash U-boot and later the kernel and filesystem in NAND flash. As far as bootloaders are concerned, the layout of the NAND flash will look like:



- Offset `0x0` for the first stage bootloader is dictated by the hardware: the ROM code of the OMAP looks for a bootloader at offset `0x0` in the NAND flash.
- Offset `0x80000` for the second stage bootloader is decided by the first stage bootloader. This can be changed by changing the U-Boot configuration.
- Offset `0x260000` of the U-Boot environment is also decided by the U-Boot configuration.

Let's first erase the whole NAND storage to remove its existing contents. This way, we are sure that what we find in NAND comes from our own manipulations:

```
nand erase.chip
```

We are going to flash the first stage bootloader in NAND. To do so, type the following commands:

```
fatload mmc 0 80000000 MLO
```

This loads the file from MMC 0 partition 0 to memory at address 0x80000000.

```
nandecc hw
```

This tells U-Boot to write data to NAND using the hardware ECC algorithm, which the ROM code of the OMAP uses to load the first stage bootloader.

```
nand erase 0 80000
```

This command erases a 0x80000 byte long space of NAND flash from offset 0[3].

```
nand write 80000000 0 80000
```

This command writes data to NAND flash. The source is 0x80000000 (where we've loaded the file to store in the flash) and the destination is offset 0 of NAND flash. The length of the copy is 0x80000 bytes, which corresponds to the space we've just erased before. It is important to erase the flash space before trying to write on it.

Now that the first stage has been transfered to NAND flash, you can now do the same with U-Boot.

The storage offset of U-Boot in the NAND is 0x80000 (just after the space reserved for the first stage bootloader) and the length is 0x1e0000.

After flashing the U-Boot image, also erase the U-boot environment variables defined by the manufacturer or by previous users of your board:

```
nand erase 260000 80000
```

You can remove MMC card, then reset the IGEP board. You should see the freshly flashed U-Boot starting.

You should now see the U-Boot prompt:

```
U-Boot #
```

Setting up Ethernet communication

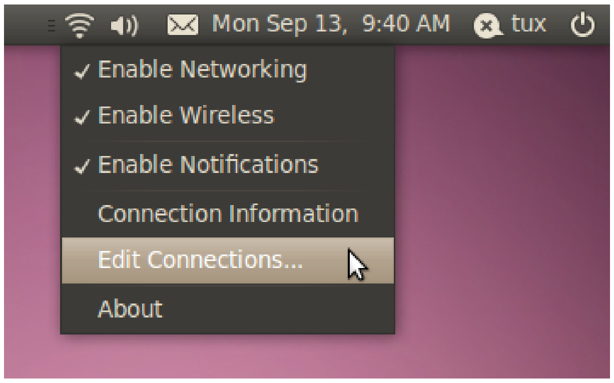
Later on, we will transfer files from the development workstation to the board using the TFTP protocol, which works on top of an Ethernet connection.

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

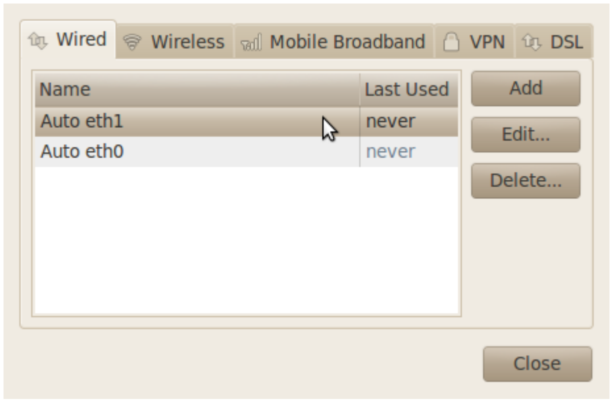
With a network cable, connect the Ethernet port of your board to the one of your computer. If your computer already has a wired connection to the network, your instructor will provide you with a USB Ethernet adapter. A new network interface, probably `eth1` or `eth2`, should appear on your Linux system.

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

[3]Of course, this is not needed here if you erased the whole NAND contents as instructed earlier. However, we prefer to write it here so that you don't forget next time you write anything to NAND.

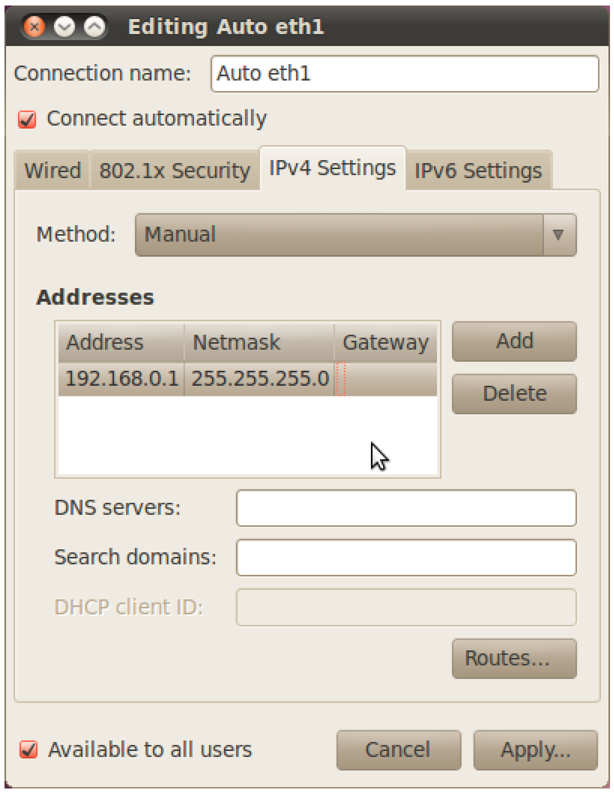


Select the new *wired network connection*:



Name	Last Used
Auto eth1	never
Auto eth0	never

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



Editing Auto eth1

Connection name: Auto eth1

✓ Connect automatically

Method: Manual

Addresses

Address	Netmask	Gateway
192.168.0.1	255.255.255.0	

DNS servers:

Search domains:

DHCP client ID:

✓ Available to all users

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

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

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

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

```
setenv ethaddr 01:02:03:04:05:06
```

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 switch your board off and on again[4].

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 0x80000000 textfile.txt
```

[4]Power cycling 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.

Caution: known issue in Ubuntu 12.04 and later: if this command doesn't work, you may have to stop the server and start it again every time you boot your workstation:

```
sudo service tftpd-hpa restart
```

The `tftp` command should have downloaded the `textfile.txt` file from your development workstation into the board's memory at location 0x80000000 (this location is part of the board DRAM). You can verify that the download was successful by dumping the contents of the memory:

```
md 0x80000000
```

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 `MLO` and `u-boot.img` files, you will find working versions under `data/` in the current lab directory.

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/felabs/sysdev/kernel` directory and go into it.

Get the sources

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

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

We will use `linux-3.11.x`, which this lab was tested with.

To practice the patch command later, download the full 3.10 sources. Unpack the archive, which creates a `linux-3.10` directory. Remember that you can use `wget <URL>` on the command line to download files.

Apply patches

Download the 2 patch files corresponding to the latest 3.11 stable release: a first patch to move from 3.10 to 3.11 and a second patch to move from 3.11 to 3.11.x.

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

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

Rename the `linux-3.10` directory to `linux-3.11.<x>`.

Kernel - Cross-compiling

Objective: Learn how to cross-compile a kernel for an OMAP 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 IGEPv2 arm board
- Use U-Boot to download the kernel
- Check that the kernel you compiled starts the system

Setup

Go to the `$HOME/felabs/sysdev/kernel` directory.

Install the following packages: `libqt4-dev` and `u-boot-tools`. `libqt4-dev` is needed for the `xconfig` kernel configuration interface, and `u-boot-tools` is needed to build the `uImage` kernel image file for U-Boot.

Target system

We are going to cross-compile and boot a Linux kernel for the IGEPv2 board.

Kernel sources

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

Cross-compiling environment setup

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

```
export PATH=/usr/local/xtools/arm-unknown-linux-uclibcgnueabi/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 IGEPv2 board (hint: the default configuration is not named after the board, but after the CPU 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:

- Disable support for the IGEPv2 board compiled into the kernel (`CONFIG_MACH_IGEP0020`). We will boot our kernel with a device tree for our board, and won't compile the board description file (`arch/arm/mach-omap2/board-igep0020.c` in the kernel sources.). Technically speaking, you can leave this option enabled, and still boot using a *Device Tree*, but disabling it makes sure that your board will not fall back to legacy booting if you do a mistake! You will have to review dependencies to be able to disable this kernel configuration setting.
- As an experiment, let's change the kernel compression from Gzip to XZ. This compression algorithm is far more efficient than Gzip, in terms of compression ratio, at the expense of a higher decompression time.

Cross compiling

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

```
make
```

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

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

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

```
make LOADADDR=0x80008000 uImage
```

The `LOADADDR` indicates to U-Boot where the kernel image should be loaded.

Setting up serial communication with the board

Plug the IGEP board on your computer. Start Picocom on `/dev/ttyS0`, or on `/dev/ttyUSB0` if you are using a serial to USB adapter.

You should now see the U-Boot prompt:

```
U-Boot #
```

Make sure that the bootargs U-Boot environment variable is not set (it could remain from a previous training session, and this could disturb the next lab):

```
setenv bootargs
saveenv
```

Load and boot the kernel using U-Boot

We will use TFTP to load the kernel image to the IGEP board:

- On your workstation, copy the `uImage` and DTB files to the directory exposed by the TFTP server.
- On the target, load `uImage` from TFTP into RAM at address 0x80000000:
  `tftp 0x80000000 uImage`
- Now, also load the DTB file into RAM at address 0x81000000:
  `tftp 0x81000000 omap3-igep0020.dtb`
- Boot the kernel with its device tree:
  `bootm 0x80000000 - 0x81000000`

You should see Linux boot and finally hang with the following message:

```
Waiting for root device /dev/mmcblk0p2...
```

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

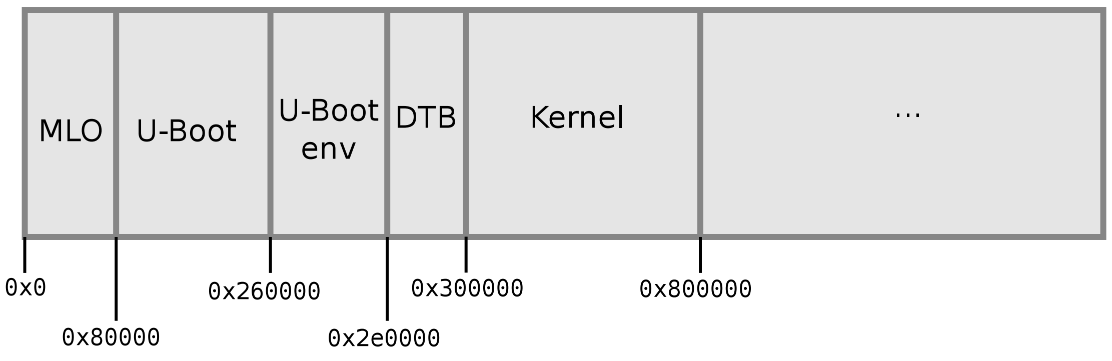
You can now automate all this every time the board is booted or reset. Reset the board, and specify a different `bootcmd`:

```
setenv bootcmd 'tftp 80000000 uImage; tftp 81000000 omap3-igep0020.dtb; bootm 80000000 - 81000000'
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 IGEP board. See the bootloader lab for details about U-boot's `nand` command.

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 0x2e0000 0x20000
        (NAND offset) (size)
```

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

```
nand erase 0x300000 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 0x81000000 0x2e0000 0x20000
           (RAM addr) (NAND offset) (size)
nand write 0x80000000 0x300000 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 0x81000000 0x2e0000 0x20000
nboot 0x80000000 0        0x300000
      (RAM addr) (device) (NAND offset)
bootm 0x80000000 - 0x81000000
```

`nboot` copies the kernel to RAM, using the `uImage` headers to know how many bytes to copy. To load the kernel to RAM, image, you could have used `nand read 0x80000000 0x300000 0x500000`, but you would have copied more bytes than the actual size of your kernel. [5].

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

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

[5] `nboot` can save a lot of boot time, as it avoids having to copy a pessimistic number of bytes from flash to RAM. Note that U-boot is not always configured with `nboot` support.

Tiny embedded system with BusyBox

Objective: making a tiny yet full featured embedded system

After this lab, you will:

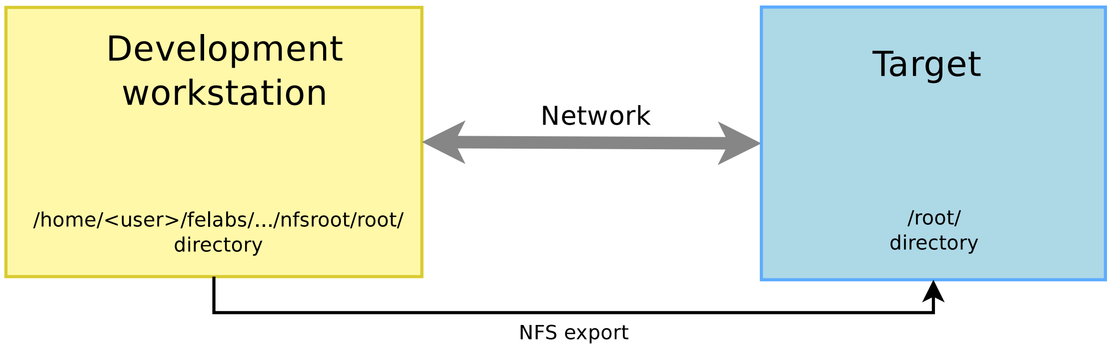
- be able to configure and build a Linux kernel that boots on a directory on your workstation, shared through the network by NFS.
- be able to create and configure a minimalistic root filesystem from scratch (ex nihilo, out of nothing, entirely hand made...) for the IGEP board
- understand how small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

Lab implementation

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

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

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



Setup

Go to the `$HOME/felabs/sysdev/tinysystem/` directory.

Kernel configuration

We will re-use the kernel sources from our previous lab, in `$HOME/felabs/sysdev/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>/felabs/sysdev/tinysystem/nfsroot 192.168.0.100(rw,no_root_squash,no_subtree_check)
```

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

Then, restart the NFS server:

```
sudo /etc/init.d/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** (Caution: in `ttyO2` below, it's the capital letter `O`, like in **O**MAP and not the number zero):

```
setenv bootargs console=ttyO2,115200 root=/dev/nfs ip=192.168.0.100
   nfsroot=192.168.0.1:/home/<user>/felabs/sysdev/tinysystem/nfsroot rw
```

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

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

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

```
[    7.467895] VFS: Mounted root (nfs filesystem) readonly on device 0:12.
```

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 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 latest BusyBox 1.21.x release.

To configure BusyBox, we won't be able to use `make xconfig`, which is currently broken in Ubuntu 12.04, because of Qt library dependencies.

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

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

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

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

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

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

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

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

Virtual filesystems

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

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

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

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

System configuration and startup

The first userspace 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.

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

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?

Switching to shared libraries

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

You will first encounter a `not found` error caused by the absence of 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. Similarly, find these libraries in the toolchain and copy them to `lib/` on the target.

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

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

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

Implement a web interface for your device

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

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

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

It will automatically background itself.

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

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

Filesystems - 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 MMC flash drive. The filesystem will be split into several partitions, and your IGEP board will be booted with this MMC card, without using NFS anymore.

Setup

Throughout this lab, we will continue to use the root filesystem we have created in the `$HOME/felabs/sysdev/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 ext3.

Boot your board with this new kernel and on the NFS filesystem you used in this previous lab.[7]

Add partitions to the MMC card

Using `cfdisk` [8], add two additional partitions to the MMC card (in addition to the existing `boot` partition created in the bootloaders lab):

- One partition, 8 MB big [9], 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 MMC card, that will be used for the data filesystem. Here also, keep the `Linux` type for the partition.

[7] If you didn't do or complete the tinysystem lab, you can use the `data/rootfs` directory instead.

[8] Now that one partition already exists, you don't have to specify headers and sectors again. Just run `cfdisk /dev/sdx`

[9] 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.

At the end, you should have three partitions: one for the boot, one for the root filesystem and one for the data filesystem.

Data partition on the MMC disk

Caution: read this carefully before proceeding. You could destroy existing partitions on your PC!
Do not make the confusion between the device that is used by your board to represent your MMC disk (`/dev/mmcblk0`) or `/dev/sda` if you are connecting the MMC card to the board with a USB card reader), and the device that your workstation uses (probably `/dev/sdb`).
So, don't use the `/dev/sdaX` device to reflash your MMC disk from your workstation. People have already destroyed their Windows partition by making this mistake.

Using the `mkfs.ext3` create a journaled file system on the third partition of the MMC disk. Remember that you can use the `-L` option to set a volume name for the partition. Move the contents of the `www/upload/files` directory (in your target root filesystem) into this new partition. The goal is to use the third partition of the MMC card as the storage for the uploaded images.

Connect the MMC disk to your board (after rebooting the board... that's currently needed with Linux 3.11 with the Device Tree supported IGEPv2 board). You should see the MMC partitions in `/proc/partitions`.

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 MMC disk 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 MMC disk.

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 MMC disk.

Booting on the SquashFS partition

In the U-boot shell, configure the kernel command line to use the second partition of the MMC disk as the root file system. Also add the `rootwait` boot argument, to wait for the MMC disk to be properly initialized before trying to mount the root filesystem. Since the MMC 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 MMC card

Finally, copy the `uImage` kernel image and DTB to the first partition of the MMC card (the partition called `boot`), and adjust the `bootcmd` of U-Boot so that it loads the kernel and DTB from the MMC card instead of loading them through the network[10].

[10]Go back to the instructions in the "Bootloader - U-Boot" lab if you don't remember how to load files from an MMC card.

Filesystems - Flash file systems

Objective: Understand 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.

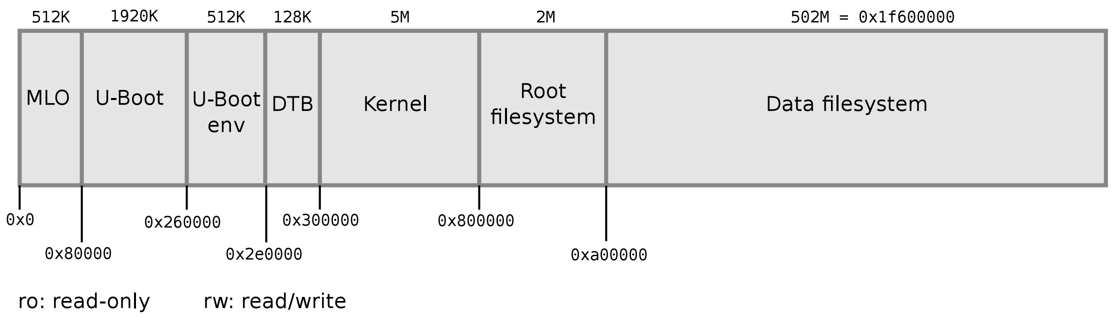
Setup

Stay in `$HOME/felabs/sysdev/tinysystem`. Install the `mtd-utils` package, which will be useful to create JFFS2 filesystem images.

Goals

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

The root filesystem will still be in a read-only filesystem, put on an MTD partition. Read/write data will be stored in a JFFS2 filesystem in another MTD partition. The layout of the internal NAND flash will be:



Enabling NAND flash and filesystems

Apply the `0001-ARM-omap-switch-back-to-SW-ECC-on-IGEP.patch` patch available in the `$HOME/felabs/sysdev/flash-filesystems/data` directory to your kernel tree. This is a Device Tree patch that fixes a problem of the 3.11.x/3.12 kernel, which caused the kernel to use a different ECC scheme than the U-Boot bootloader, making it impossible to flash from U-Boot, and read from the kernel.

After applying this patch, recompile your kernel with support for JFFS2 and for support for MTD partitions specified in the kernel command line (`CONFIG_MTD_CMDLINE_PARTS`).

Also enable support for the flash chips on the board (`CONFIG_MTD_NAND_OMAP2`). You also need to enable support for hardware BCH error correction (`CONFIG_NAND_OMAP_BCH`) and select the `8 bits / 512 bytes (recommended)` mode (`MTD_NAND_OMAP_BCH8`).

Last but not least, disable `CONFIG_PROVE_LOCKING`. This option is currently causing problems with the JFFS2 filesystem. This option is in `Kernel Hacking` → `Lock debugging: prove locking correctness`.

After compiling your kernel, update the DTB file used by your board (remember that we applied a Device Tree patch).

You will update your kernel image on flash in the next section.

Filesystem image preparation

Find the erase block size of the NAND flash device in your board.

Prepare a JFFS2 filesystem image from the `/www/upload/files` directory from the previous lab.

Modify the `/etc/init.d/rcS` file to mount a JFFS2 filesystem on the seventh flash partition (we will declare flash partitions in the next section), instead of an ext3 filesystem on the third MMC disk partition.

Create a JFFS2 image for your root filesystem, with the same options as for the data filesystem.

MTD partitioning and flashing

Look at the way default flash partitions are defined in the board Device Tree sources (`arch/arm/boot/dts/omap3-igep0020.dts`).

However, 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.

Enter the U-Boot shell and erase NAND flash, from offset 0x300000, up to the end of the NAND flash storage. You'll have to compute the remaining size of the flash, from 0x300000 to the end. Remember that you can look at U-Boot booting messages to find what the size of the NAND flash is.

Before flashing JFFS2 images, make sure they will be flashed during the software ECC scheme, by running the `nandecc sw` command in U-Boot.

Using the `tftp` command, download and flash the new kernel image at the correct location.

Using the `tftp` command, download and flash the JFFS2 image of the root filesystem the correct location.

Using the `tftp` command, download and flash the JFFS2 image of the data filesystem at the correction location.

Don't forget that you can write U-Boot scripts to automate these procedures. This is very handy to avoid mistakes when typing commands!

Set the `bootargs` variable so that:

- You define the 7 MTD partitions, as detailed previously
- The root filesystem is mounted from the 6th partition, and is mounted read-only (kernel parameter `ro`). **Important: even if this partition is mounted read-only, the MTD partition itself must be declared as read-write. Otherwise, Linux won't be able to perform ECC checks on it, which involve both reading and writing.**

Boot the target, check that MTD partitions are well configured, and 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.

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 *DirectFB* graphic library and sample applications using this library. Because many boards do not have a display, we will test the result of this lab with *QEMU*.

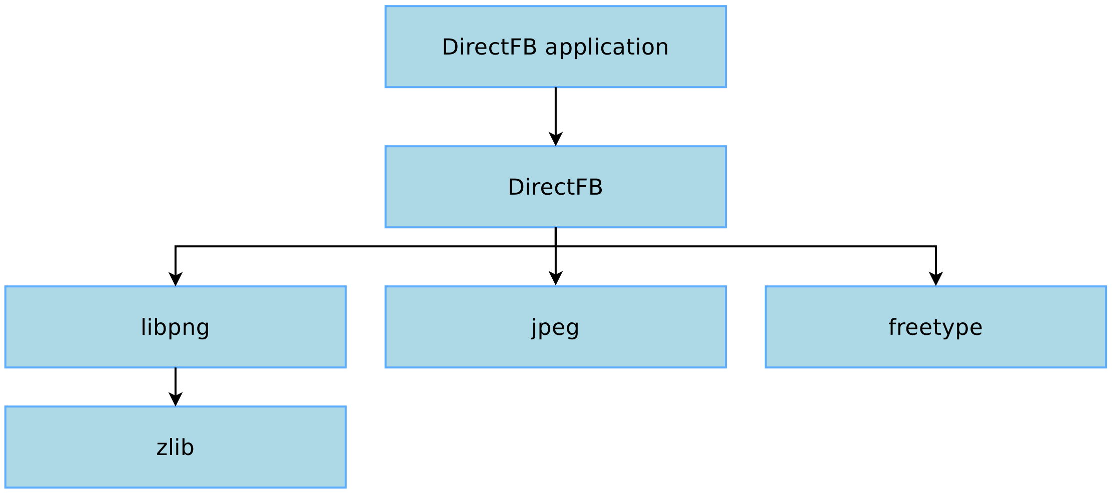
We'll see that manually re-using existing libraries is quite tedious, so that more automated procedures are necessary to make it easier. However, learning how to perform these operations manually will significantly help you when you'll face issues with more automated tools.

Figuring out library dependencies

As most libraries, DirectFB depends on other libraries, and these dependencies are different depending on the configuration chosen for DirectFB. In our case, we will enable support for:

- PNG image loading
- JPEG image loading
- Font rendering using a font engine

The PNG image loading feature will be provided by the *libpng* library, the JPEG image loading feature by the *jpeg* library and the font engine will be implemented by the *FreeType* library. The *libpng* library itself depends on the *zlib* compression/decompression library. 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 *zlib*, then continue with *libpng*, *jpeg* and *FreeType*, to finally compile *DirectFB* and the *DirectFB* sample applications.

Preparation

For our cross-compilation work, we will need to 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 copy only 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/felabs/sysdev/thirdparty`, create two directories: `staging` and `target`.

For the target, we need a basic system with BusyBox, device nodes 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:

```
sudo cp -a $HOME/felabs/sysdev/tinysystem/nfsroot/* target/
```

The copy must be done as `root`, because the root filesystem of the *A tiny embedded system* lab contains a few device nodes.

Testing

Make sure the `target/` directory is exported by your NFS server by adding the following line to `/etc/exports`:

```
/home/<user>/felabs/sysdev/thirdparty/target 172.20.0.2(rw,no_root_squash,no_subtree_check)
```

And restart your NFS server.

Install the QEMU emulator for non-x86 architectures by installing the `qemu-kvm-extras` package.

Then, run QEMU with the provided script:

```
./run_qemu
```

The system should boot and give you a prompt.

zlib

`Zlib` is a compression/decompression library available at `http://www.zlib.net/`. Download version 1.2.5, and extract it in `$HOME/felabs/sysdev/thirdparty/`.

By looking at the `configure` script, we see that this configure script has not been generated by `autoconf` (otherwise it would contain a sentence like *Generated by GNU Autoconf 2.62*). Moreover, the project doesn't use automake since there are no Makefile.am files. So zlib uses a custom build system, not a build system based on the classical autotools.

Let's try to configure and build zlib:

```
./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 tell the configure script to use the ARM cross-compiler:

```
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. Moreover, the beginning of the configure script tells us about this:

```
# To impose specific compiler or flags or
# install directory, use for example:
#    prefix=$HOME CC=cc CFLAGS="-O4" ./configure
```

Now when you compile with make, the cross-compiler is used. Look at the result of compiling: a set of object files, a file `libz.a` and set of `libz.so*` files.

The `libz.a` file is the static version of the library. It has been generated using the following command:

```
ar rc libz.a adler32.o compress.o crc32.o gzio.o uncompr.o deflate.o \
      trees.o zutil.o inflate.o infback.o inftrees.o inffast.o
```

It can be used to compile applications linked statically with the zlib library, as shown by the compilation of the example program:

```
arm-linux-gcc -O3 -DUSE_MMAP -o example example.o -L. libz.a
```

In addition to this static library, there is also a dynamic version of the library, the `libz.so*` files. The shared library itself is `libz.so.1.2.5`, it has been generated by the following command line:

```
arm-linux-gcc -shared -Wl,-soname,libz.so.1 -o libz.so.1.2.5 \
              adler32.o compress.o crc32.o gzio.o uncompr.o  \
              deflate.o trees.o zutil.o inflate.o infback.o  \
              inftrees.o inffast.o
```

And creates symbolic links `libz.so` and `libz.so.1`:

```
ln -s libz.so.1.2.5 libz.so
ln -s libz.so.1.2.5 libz.so.1
```

These symlinks are needed for two different reasons:

- `libz.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 `-lz` and the name of the library file is `libz.so`. So, the `libz.so` symlink is needed at compile time;

- `libz.so.1` 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 libz.so.1.2.5
```

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 zlib 1.2.5, and fixed in the next release 1.2.6. You can recompile the library, install it on your target system, change the link `libz.so.1` so that it points to `libz.so.1.2.6` and restart your applications. And it will work, because your applications don't look specifically for `libz.so.1.2.5` but for the *SONAME* `libz.so.1`. However, it also means that as a library developer, if you break the ABI of the library, you must change the *SONAME*: change from `libz.so.1` to `libz.so.2`.

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 --prefix=/usr
make
```

For the zlib 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, zlib will be installed in `$HOME/felabs/sysdev/thirdparty/target/usr/lib/` because this is the directory where we are building the root filesystem, but once our target system will be running, it will see zlib in `/usr/lib`. The prefix corresponds to the path in the target system and **never** on the host. So, one should **never** pass a prefix like `$HOME/felabs/sysdev/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 overriden, and be different from the configuration prefix.

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

```
make DESTDIR=../staging install
```

Now look at what has been installed by zlib:

- A manpage in `/usr/share/man`
- A pkgconfig file in `/usr/lib/pkgconfig`. We'll come back to these later
- The shared and static versions of the library in `/usr/lib`
- The headers in `/usr/include`

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 `libz.so.1` and `libz.so.1.2.5` are needed, since `libz.so.1` is the *SONAME* of the library and `libz.so.1.2.5` is the real binary:
   `cp -a libz.so.1* ../target/usr/lib`
3. Strip the library:
   `arm-linux-strip ../target/usr/lib/libz.so.1.2.5`

Ok, we're done with zlib!

Libpng

Download libpng from its official website at `http://www.libpng.org/pub/png/libpng.html`. We tested the lab with version 1.4.3. Please stick to this version as newer versions are incompatible with the DirectFB version we use in this lab.

Once uncompressed, we quickly discover that the libpng build system is based on the *autotools*, so we will work with a regular configure script.

As we've seen previously, if we just run `./configure`, the build system will use the native compiler to build the library, which is not what we want. So let's tell the build system to use the cross-compiler:

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

Quickly, you should get an error saying:

```
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 config.log, you quickly understand what's going on:
configure:2942: checking for C compiler default output file name
configure:2964: arm-linux-gcc    conftest.c  >&5
configure:2968: $? = 0
configure:3006: result: a.out
configure:3023: checking whether the C compiler works
configure:3033: ./a.out
./configure: line 3035: ./a.out: cannot execute binary file
```

The configure script compiles a binary with the cross-compiler and then tries to run it on the development workstation. Obviously, it cannot work, and the system says that it `cannot execute binary file`. 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
```

Now, we go a little bit further in the execution of the configure script, until we reach:

```
checking for zlibVersion in -lz... no
configure: error: zlib not installed
```

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

```
configure:12452: checking for zlibVersion in -lz
configure:12487: arm-linux-gcc -o conftest -g -O2   conftest.c -lz  -lm  >&5
/usr/local/xtools/arm-unknown-linux-uclibcgnueabi/[...]usr/bin/[...]/ld: cannot find -lz
collect2: ld returned 1 exit status
```

The configure script tries to compile an application against *zlib* (as can be seen from the `-lz` option): *libpng* uses the *zlib* library, so the `configure` script wants to make sure this library is already installed. Unfortunately, the `ld` linker doesn't find this library. 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/felabs/sysdev/thirdparty/staging/usr/lib \
CC=arm-linux-gcc \
./configure --host=arm-linux
```

Let's also specify the prefix, so that the library is compiled to be installed in `/usr` and not `/usr/local`:

```
 LDFLAGS=-L$HOME/felabs/sysdev/thirdparty/staging/usr/lib \
 CC=arm-linux-gcc \
./configure --host=arm-linux --prefix=/usr
```

Then, run the compilation using make. Quickly, you should get a pile of error messages, starting with:

```
In file included from png.c:13:
png.h:470:18: error: zlib.h: No such file or directory
```

Of course, since *libpng* uses the *zlib* library, 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<includedir> if you have headers
                        in a nonstandard directory <includedir>
```

Let's use it:

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

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/libpng/ 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, the files will be installed in `DESTDIR/configuration-prefix`.

Now, let's see what has been installed in `/tmp/libpng/`:

```
./usr/lib/libpng.la                   -> libpng14.la
./usr/lib/libpng14.a
./usr/lib/libpng14.la
./usr/lib/libpng14.so                 -> libpng14.so.14.3.0
./usr/lib/libpng14.so.14              -> libpng14.so.14.3.0
./usr/lib/libpng14.so.14.3.0
./usr/lib/libpng.a                    -> libpng14.a
./usr/lib/libpng.la                   -> libpng14.la
./usr/lib/libpng.so                   -> libpng14.so
./usr/lib/pkgconfig/libpng.pc         -> libpng14.pc
./usr/lib/pkgconfig/libpng14.pc
./usr/share/man/man5/png.5
./usr/share/man/man3/libpngpf.3
./usr/share/man/man3/libpng.3
./usr/include/pngconf.h               -> libpng14/pngconf.h
./usr/include/png.h                   -> libpng14/png.h
./usr/include/libpng14/pngconf.h
./usr/include/libpng14/png.h
./usr/bin/libpng-config               -> libpng14-config
./usr/bin/libpng14-config
```

So, we have:

- The library, with many symbolic links
  - `libpng14.so.14.3.0`, the binary of the current version of library
  - `libpng14.so.14`, a symbolic link to `libpng14.so.14.3.0`, so that applications using `libpng14.so.14` as the *SONAME* of the library will find nit and use the current version
  - `libpng14.so` is a symbolic link to `libpng14.so.14.3.0`. So it points to the current version of the library, so that new applications linked with `-lpng14` will use the current version of the library `libpng.so` is a symbolic link to libpng14.so. So applications linked with `-lpng` will be linked with the current version of the library (and not the obsolete one since we don't want anymore to link applications against the obsolete version!)
  - `libpng14.a` is a static version of the library
  - `libpng.a` is a symbolic link to `libpng14.a`, so that applications statically linked with `libpng.a` will in fact use the current version of the library

  - `libpng14.la` is a configuration file generated by *libtool* which gives configuration details for the library. It will be used to compile applications and libraries that rely on libpng.
  - `libpng.la` is a symbolic link to `libpng14.la`: we want to use the current version for new applications, once again.
- The *pkg-config* files, in `/usr/lib/pkgconfig/`. These configuration files are used by the pkg-config tool that we will cover later. They describe how to link new applications against the library.
- The manual pages in `/usr/share/man/`, explaining how to use the library.
- The header files, in `/usr/include/`, needed to compile new applications or libraries against *libpng*. They define the interface to *libpng*. There are symbolic links so that one can choose between the following solutions:
  - Use `#include <png.h>` in the source code and compile with the default compiler flags
  - Use `#include <png.h>` in the source code and compile with `-I/usr/include/libpng14`
  - Use `#include <libpng14/png.h>` in the source and compile with the default compiler flags
- The `/usr/bin/libpng14-config` tool and its symbolic link `/usr/bin/libpng-config`. This tool is a small shell script that gives configuration information about the libraries, needed to know how to compile applications/libraries against libpng. This mechanism based on shell scripts is now being superseded by *pkg-config*, but as old applications or libraries may rely on it, it is kept for compatibility.

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

```
make DESTDIR=$HOME/felabs/sysdev/thirdparty/staging/ install
```

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

```
cd ..
cp -a staging/usr/lib/libpng14.so.* target/usr/lib
arm-linux-strip target/usr/lib/libpng14.so.14.3.0
```

And we're finally done with libpng!

libjpeg

Now, let's work on *libjpeg*. Download it from http://www.ijg.org/files/jpegsrc.v8.tar.gz and extract it.

Configuring *libjpeg* 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/felabs/sysdev/thirdparty/staging/ install
```

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

```
cd ..
cp -a staging/usr/lib/libjpeg.so.8* target/usr/lib/
arm-linux-strip target/usr/lib/libjpeg.so.8.0.0
```

Done with libjpeg!

FreeType

The *FreeType* library is the next step. Grab the tarball from http://www.freetype.org. We tested the lab with version 2.4.2 but more other versions may also work. Uncompress the tarball.

The FreeType build system is a nice example of what can be done with a good usage of the autotools. Cross-compiling FreeType is very easy. First, the configure step:

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

Then, compile the library:

```
make
```

Install it in the *staging* space:

```
make DESTDIR=$HOME/felabs/sysdev/thirdparty/staging/ install
```

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

```
cd ..
cp -a staging/usr/lib/libfreetype.so.6* target/usr/lib/
arm-linux-strip target/usr/lib/libfreetype.so.6.6.0
```

Done with FreeType!

DirectFB

Finally, with *zlib*, *libpng*, *jpeg* and *FreeType*, all the dependencies of DirectFB are ready. We can now build the DirectFB library itself. Download it from the official website, at http://www.directfb.org/. We tested version 1.4.5 of the library. As usual, extract the tarball.

Before configuring DirectFB, let's have a look at the available options by running `./configure --help`. A lot of options are available. We see that:

- Support for Fbdev (the Linux framebuffer) is automatically detected, so that's fine;
- Support for PNG, JPEG and FreeType is enabled by default, so that's fine;
- We should specify a value for `--with-gfxdrivers`. The hardware emulated by QEMU doesn't have any accelerated driver in DirectFB, so we'll pass `--with-gfxdrivers=none`;
- We should specify a value for `--with-inputdrivers`. We'll need keyboard (for the keyboard) and linuxinput to support the Linux Input subsystem. So we'll pass `--with-inputdrivers=keyboard,linuxinput`

So, let's begin with a configure line like:

```
CC=arm-linux-gcc ./configure --host=arm-linux \
        --prefix=/usr --with-gfxdrivers=none \
        --with-inputdrivers=keyboard,linuxinput
```

In the output, we see:

```
*** JPEG library not found. JPEG image provider will not be built.
```

So let's look in config.log for the JPEG issue. By search for `jpeg`, you can find:

```
configure:24701: arm-linux-gcc -o conftest [...] conftest.c -ljpeg  -ldl -lpthread  >&5
/usr/local/xtools/arm-unknown-linux-uclibcgnueabi/[...]/bin/ld: cannot find -ljpeg
```

Of course, it cannot find the jpeg library, since we didn't pass the proper `LDFLAGS` and `CFLAGS` telling where our libraries are. So let's configure again with:

```
LDFLAGS=-L$HOME/felabs/sysdev/thirdparty/staging/usr/lib \
CPPFLAGS=-I$HOME/felabs/sysdev/thirdparty/staging/usr/include \
CC=arm-linux-gcc \
./configure --host=arm-linux --prefix=/usr \
 --with-gfxdrivers=none --with-inputdrivers=keyboard,linuxinput
```

Ok, now at the end of the configure, we get:

```
JPEG                yes  -ljpeg
PNG                 yes  -I/usr/include/libpng12 -lpng12
[...]
FreeType2           yes  -I/usr/include/freetyp2 -lfreetype
```

It found the JPEG library properly, but for libpng and freetype, it has added `-I` options that points to the libpng and freetype libraries installed on our host (x86) and not the one of the target. This is not correct!

In fact, the DirectFB configure script uses the *pkg-config* system to get the configuration parameters to link the library against libpng and FreeType. By default, *pkg-config* looks in `/usr/lib/pkgconfig/` for `.pc` files, and because the `libfreetype6-dev` and `libpng12-dev` packages are already installed in your system (it was installed in a previous lab as a dependency of another package), then the configure script of DirectFB found the libpng and FreeType libraries of your host!

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. In our case, if `libfreetype6-dev` was not installed, then the `/usr/lib/pkgconfig/freetype2.pc` file wouldn't exist, and the configure script of DirectFB would have said something like *Sorry, can't find FreeType*.

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_PATH` environment variable, as explained in the manual page of `pkg-config`.

Moreover, the `.pc` files contain references to paths. For example, in `$HOME/felabs/sysdev/thirdparty/staging/usr/lib/pkgconfig/freetype2.pc`, we can see:

```
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include
[...]
Libs: -L${libdir} -lfreetype
Cflags: -I${includedir}/freetype2 -I${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 DirectFB again, passing the `PKG_CONFIG_PATH` and `PKG_CONFIG_SYSROOT_DIR` environment variables:

```
LDFLAGS=-L$HOME/felabs/sysdev/thirdparty/staging/usr/lib \
CPPFLAGS=-I$HOME/felabs/sysdev/thirdparty/staging/usr/include \
PKG_CONFIG_PATH=$HOME/felabs/sysdev/thirdparty/staging/usr/lib/pkgconfig \
PKG_CONFIG_SYSROOT_DIR=$HOME/felabs/sysdev/thirdparty/staging \
CC=arm-linux-gcc \
./configure --host=arm-linux --prefix=/usr \
 --with-gfxdrivers=none --with-inputdrivers=keyboard,linuxinput
```

Ok, now, the lines related to Libpng and FreeType 2 looks much better:

```
PNG       yes -I/home/<user>/felabs/sysdev/thirdparty/staging/usr/include/libpng14 -lpng14
FreeType2 yes -I/home/<user>/felabs/sysdev/thirdparty/staging/usr/include/freetype2 -lfreetype
```

Let's build DirectFB with make. After a while, it fails, complaining that `X11/Xlib.h` and other related header files cannot be found. In fact, if you look back the `./configure` script output, you can see:

```
X11 support yes -lX11 -lXext
```

Because X11 was installed on our host, DirectFB `./configure` script thought that it should enable support for it. But we won't have X11 on our system, so we have to disable it explicitly. In the `./configure --help` output, one can see:

```
--enable-x11 build with X11 support [default=auto]
```

So we have to run the configuration again with the same arguments, and add `--disable-x11` to them.

The build now goes further, but still fails with another error:

```
/usr/lib/libfreetype.so: could not read symbols: File in wrong format
```

As you can read from the above command line, the Makefile is trying to feed an x86 binary (`/usr/lib/libfreetype.so`) to your ARM toolchain. Instead, it should have been using `usr/lib/libfreetype.so` found in your staging environment.

This happens because the *libtool* `.la` files in your staging area need to be fixed to describe the right paths in this staging area. So, in the .la files, replace `libdir='/usr/lib'` by `libdir='/home/<user>/felabs/sysdev/thirdparty/staging/usr/lib'`. Restart the build again, preferably from scratch (`make clean` then `make`) to be sure everything is fine.

Finally, it builds!

Now, install DirectFB to the *staging* space using:

```
make DESTDIR=$HOME/felabs/sysdev/thirdparty/staging/ install
```

And so the installation in the *target* space:

- First, the libraries:

  ```
  cd ..
  cp -a staging/usr/lib/libdirect-1.4.so.5* target/usr/lib
  cp -a staging/usr/lib/libdirectfb-1.4.so.5* target/usr/lib
  cp -a staging/usr/lib/libfusion-1.4.so.5* target/usr/lib
  arm-linux-strip target/usr/lib/libdirect-1.4.so.5.0.0
  arm-linux-strip target/usr/lib/libdirectfb-1.4.so.5.0.0
  arm-linux-strip target/usr/lib/libfusion-1.4.so.5.0.0
  ```

- Then, the plugins that are dynamically loaded by DirectFB. We first copy the whole `/usr/lib/directfb-1.4-5/` directory, then remove the useless files (`.la`) and finally strip the `.so` files:

```
cp -a staging/usr/lib/directfb-1.4-5/ target/usr/lib
find target/usr/lib/directfb-1.4-5/ -name '*.la' -exec rm {} \;
find target/usr/lib/directfb-1.4-5/ -name '*.so' -exec arm-linux-strip {} \;
```

DirectFB examples

To test that our DirectFB installation works, we will use the example applications provided by the DirectFB project. Start by downloading the tarball at http://www.directfb.org/downloads/Extras/DirectFB-examples-1.2.0.tar.gz and extract it.

Then, we configure it just as we configured DirectFB:

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

Then, compile it with `make`. Soon a compilation error will occur because `bzero` is not defined. The `bzero` function is a deprecated BSD function, and `memset` should be used instead. The GNU C library still defines `bzero`, but by default, the uClibc library doesn't provide `bzero` (to save space). So, let's modify the source code in `src/df_knuckles/matrix.c` to change the line:

```
#define M_CLEAR(m) bzero(m, MATRIX_SIZE)
```

to

```
#define M_CLEAR(m) memset(m, 0, MATRIX_SIZE)
```

Run the compilation again, it should succeed.

For the installation, as DirectFB examples are only applications and not libraries, we don't really need them in the *staging* space, but only in the *target* space. So we'll directly install in the *target* space using the `install-strip` make target. This make target is usually available with autotools based build systems. In addition to the destination directory (`DESTDIR` variable), we must also tell which strip program should be used, since stripping is an architecture-dependent operation (`STRIP` variable):

```
make STRIP=arm-linux-strip \
     DESTDIR=$HOME/felabs/sysdev/thirdparty/target/ install-strip
```

Final setup

Start the system in QEMU using the `run_qemu` script, and try to run the `df_andi` program, which is one of the DirectFB examples.

The application will fail to run, because the *pthread* library (which is a component of the C library) is missing. This library is available inside the toolchain. So let's add it to the target:

```
TOOLCHAIN_SYSROOT=$(arm-linux-gcc -print-sysroot)
cp -a $TOOLCHAIN_SYSROOT/lib/libpthread* target/lib/
```

Then, try to run `df_andi` again. It will complain about `libdl`, which is used to dynamically load libraries during application execution. So let's add this library to the target:

```
cp -a $TOOLCHAIN_SYSROOT/lib/libdl* target/lib
```

When running `df_andi` again, it will complain about `libgcc_s`, so let's copy this library to the target:

```
cp -a $TOOLCHAIN_SYSROOT/lib/libgcc_s* target/lib
```

Now, the application should no longer complain about missing libraries. But when started, it should complain about the `/dev/fb0` device file that doesn't exist. So let's create this device file:

```
sudo mknod target/dev/fb0 c 29 0
```

Next executing the application will complain about missing `/dev/ttyx` device files, so let's create them:

```
sudo mknod target/dev/tty0 c 4 0
sudo mknod target/dev/tty5 c 4 5
```

Finally, when running `df_andi`, another error message shows up:

```
Unable to dlopen '/usr/lib/[...]/libidirectfbimageprovider_png.so' !
File not found
```

DirectFB is trying to load the PNG plugin using the `dlopen()` function, which is part of the `libdl` library we added to the target system before. Unfortunately, loading the plugin fails with the *File not found* error. However, the plugin is properly present, so the problem is not the plugin itself. What happens is that the plugin depends on the *libpng* library, which itself depends on the *mathematic* library. And the mathematic library `libm` (part of the C library) has not yet been added to our system. So let's do it:

```
cp -a $TOOLCHAIN_SYSROOT/lib/libm* target/lib
```

Now, you can try and run the `df_andi` application!

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 inside Qemu.

Setup

Go to the `$HOME/felabs/sysdev/buildroot/` directory, which already contains some data needed for this lab, including a kernel image.

Get Buildroot and explore the source code

The official Buildroot website is available at `http://buildroot.org/`. Download the latest stable 2013.08.x 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 userspace 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, DirectFB and DirectFB sample applications in our embedded Linux system;
- Integrate the target filesystem into both an ext2 filesystem image and a tarball

To run the configuration utility of Buildroot, simply run:

```
make menuconfig
```

Set the following options:

- `Target Architecture`: `ARM (little endian)`
- `Target Architecture Variant`: `arm926t` (we will start booting the generated filesystem on an emulated arm9 based system, instead of the IGEPv2 board)
- `Toolchain`
  - `Toolchain type`: `External toolchain`
  - `Toolchain`: `Custom toolchain`
  - `Toolchain path`: use the toolchain you built: `/usr/local/xtools/arm-unknown-linux-uclibcgnueabi`
  - `External toolchain C library`: `uClibc`
  - We must tell Buildroot about our toolchain configuration, so: enable `Toolchain has large file support?`, `Toolchain has RPC support?`, and `Toolchain has C++ support?`. Buildroot will check these parameters anyway.
- `System configuration`
  - `Port to run a getty (login prompt) on`: change `ttyS0` to `tty1`
- `Target packages`
  - Keep `BusyBox` (default version) and keep the Busybox configuration proposed by Buildroot;
  - `In Graphic libraries and applications (graphic/text)`
    - Select `directfb`. Buildroot will automatically select the necessary dependencies.
      - Remove `enable touchscreen support`
      - Select `directfb examples`
      - Select all the DirectFB examples
- `Filesystem images`
  - Select `ext2/3/4 root filesystem`
  - 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 extract, 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 an ext2 filesystem image and a tarball of the filesystem, but depending on the Buildroot configuration, there could also be a kernel image or a bootloader image. This is where we find `rootfs.tar` and `rootfs.ext2`, which are respectively the tarball and the ext2 image of the generated root filesystem.
- `staging`, which contains the build space of the target system. All the target libraries, with headers, 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.
- `toolchain`, is the location where the toolchain is built. However, in our configuration, we re-used an existing toolchain, so this directory contains almost nothing.

Run the generated system

If you didn't do it in the previous lab, install QEMU emulator for non x86 targets:

```
sudo apt-get install qemu-kvm-extras
```

We will use the kernel image in `data` and the filesystem image generated by Buildroot in the ext2 format to boot the generated system in QEMU. We start by using a QEMU emulated ARM board with display support, allowing to test graphical applications relying on the DirectFB library. Later, we will be able move to a real board if your hardware also has a graphical display.

Execute the `run_qemu` script, which contains what's needed to boot the system in QEMU.

Log in (`root` account, no password), and run demo programs:

```
df_andi
df_dok
df_fire
```

...

Going further

- Add dropbear (SSH server and client) to the list of packages built by Buildroot, add the network emulation in QEMU (see the `../thirdparty/run_qemu` script for an example), and log to your target system in QEMU using a 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 in QEMU.

Application development

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

Setup

Go to the `$HOME/felabs/sysdev/appdev` directory.

Compile your own application

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

First, instead of using an `ext2` image, we will mount the root filesystem over NFS to make it easier to test our application. So, create a `qemu-rootfs/` directory, and inside this directory, uncompress the tarball generated by Buildroot in the previous lab (in the `output/images/` directory). Don't forget to extract the archive as `root` since the archive contains device files.

Then, run the `run_qemu` script and check that the system works as expected.

Now, our application. In the lab directory the file `data/app.c` contains a very simple DirectFB application that displays the `data/background.png` image for five seconds. 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 those 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=$HOME/felabs/sysdev/buildroot/buildroot-XXXX.YY/output/host/usr/bin:$PATH
```

Let's try to compile the application:

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

It complains that it cannot find the directfb.h header. This is normal, since we didn't tell the compiler where to find it. 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 DirectFB:[11]

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

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

[11] 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/felabs/sysdev/debugging` directory.

Debugging setup

Boot your ARM board over NFS on the filesystem produced in the *Tiny embedded system* lab, with the same kernel.

Setting up gdbserver and strace

`gdbserver` and `strace` have already been compiled for your target architecture as part of the cross-compiling toolchain. Find them in the installation directory of your toolchain. Copy these binaries to the `/usr/bin/` directory in the root filesystem of your target system.

Enabling job control

In this lab, we are going to run a buggy program that keeps hanging and crashing. Because of this, we are going to need job control, in particular `[Ctrl] [C]` allowing to interrupt a running program.

At boot time, you probably noticed that warning that job control was turned off:

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

This happens when the shell is started in the console. The system console cannot be used as a controlling terminal.

The fix is to start this shell in ttyO2 (the 3rd OMAP serial port on the IGEPv2 board) by modifying the `/etc/inittab` file:

Replace

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

which implied the use of the system console device by

```
ttyO2::askfirst:/bin/sh
```

to tell the `init` program to start the shell on `/dev/ttyO2`

Now reboot. You should no longer see the `Job control turned off` warning, and should be able to use `[Ctrl] [C]`.

Using strace

`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.

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 (you might need to create this directory if it doesn't exist yet).

```
arm-linux-gcc -o vista-emulator data/vista-emulator.c
cp vista-emulator path/to/root/filesystem/root
```

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 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 GDB `sysroot` variable (on one line):

```
(gdb) set sysroot /usr/local/xtools/arm-unknown-linux-uclibcgnueabi/
arm-unknown-linux-uclibcgnueabi/sysroot/
```

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...

- Compiling an application for the target system is very similar to compiling an application for the host, except that the cross-compilation introduces a few complexities when libraries are used.
- 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 (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 and a kernel with Xenomai.

Setup

Go to the `$HOME/felabs/realtime/rttest` directory.

If you are using a 64 bit installation of Ubuntu, install support for executables built with a 32 bit C library:

```
sudo apt-get install ia32-libs
```

This will be needed to use the toolchain from Code Sourcery.

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 2013.02 sources. Apply the Buildroot patch `buildroot-2013.02-bump-xenomai-to-2.6.2.1.patch` from the lab *data* directory to your Buildroot sources. It upgrades the Xenomai version to 2.6.2.1, which allows to use the 3.5 kernel. Apply the `0001-ext-toolchain-wrapper-fix-paths-if-executable-was-re.patch` patch from this buildroot lab's `data` directory. It fixes a bug in Buildroot's external toolchain logic.

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

- `Target architecture`: `ARM (little endian)`

- `Target Architecture Variant`: `cortex-a8`
- In `Toolchain`:
  - `Toolchain type`: `External toolchain`
  - `Toolchain`: `Sourcery CodeBench ARM 2012.03`
- In `System configuration`:
  - `/dev management`: `Dynamic using devtmpfs only`
  - `Port to run a getty (login prompt) on`: `ttyO2`
- In `Package Selection for the target`:
  - 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` is enabled.

Now, build your root filesystem.

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
```

Compile a standard Linux kernel

Download the exact Linux 3.5.7 version. That's the most recent ARM Linux version that Xenomai 2.6.2.1 supports. You will have trouble applying Xenomai kernel patches otherwise.

Apply the `linux-3.5.7-igepv2-fix-pinmux.patch` patch from this lab's `data` directory.

Configure your kernel with the default configuration for the IGEPv2 board.

In the kernel configuration interface:

- Enable `CONFIG_DEVTMPFS` and `CONFIG_DEVTMPFS_MOUNT` The root filesystem that we use has an empty `/dev` directory, and we let the kernel populate it with the devices present on the system.
- For the moment, remove `CONFIG_HIGH_RES_TIMERS`, to start by testing the kernel without high-resolution timers.
- Disable `CONFIG_SMP`, as Xenomai 2.6.1 does not support yet multi-processing on OMAP (and the IGEPv2 is anyway a single core processor).
- Disable `CONFIG_PROVE_LOCKING`, `CONFIG_DEBUG_LOCK_ALLOC`, `CONFIG_DEBUG_MUTEXES` and `CONFIG_DEBUG_SPINLOCK`.

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

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=$HOME/felabs/realtime/rttest/buildroot-2013.02/output/host/usr/bin:$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`.

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

Recompile your kernel, boot your IGEP board 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 IGEP board (where 192.168.0.100 is the IP address of the IGEP 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.

Testing Xenomai scheduling latency

Prepare the kernel for Xenomai compilation:

```
cd $HOME/felabs/realtime/rttest/buildroot-2013.02/
./output/build/xenomai-2.6.2.1/scripts/prepare-kernel.sh \
   --arch=arm --linux=/path/to/linux-3.5.7
```

Now, run 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`

In order 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/felabs/realtime/rttest/buildroot-2013.02/
make host-pkgconf
```

Compile your kernel, and in the meantime, compile `rttest` for the Xenomai POSIX skin:

```
cd $HOME/felabs/realtime/rttest/nfsroot/root
export PATH=$HOME/felabs/realtime/rttest/buildroot-2013.02/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.

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
```

Backing up your lab files

Objective: clean up and make an archive of your lab directory

End of the training session

Congratulations. You reached the end of the training session. You now have plenty of working examples you created by yourself, and you can build upon them to create more elaborate things.

In this last lab, we will create an archive of all the things you created. We won't keep everything though, as there are lots of things you can easily retrieve again.

Create a lab archive

Go to the directory containing your felabs directory:

```
cd $HOME
```

Now, run a command that will do some clean up and then create an archive with the most important files:

- Kernel configuration files
- Other source configuration files (BusyBox, Crosstool-ng...)
- Kernel images
- Toolchain
- Other custom files

Here is the command:

```
./felabs/archive-labs
```

At end end, you should have a `felabs-<user>.tar.xz` archive that you can copy to a USB flash drive, for example. This file should only be a few hundreds of MB big.