Embedded Linux training
Lab book

Free Electrons
http://free-electrons.com

About this document

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

It is is composed of several source documents that we aggregate for each training session. These individual source documents can be found on http://free-electrons.com/docs.

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

Copying this document

Corrections, suggestions, contributions and translations are welcome!

Training setup

Download files and directories used in practical labs

Update your system

Time might have elapsed since your system was last updated.

Keep your system up to date:

```
sudo apt-get update
sudo apt-get dist-upgrade
```

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 the tarball at http://free-electrons.com/labs/embedded_linux.tar.bz2.

Then, from a terminal, extract the tarball using the following command:

```
cd      (going to your home directory)
sudo tar jxf embedded_linux.tar.bz2
sudo chown -R <user>.<user> felabs
```

`root` permissions are required to extract the character and block device files contained in the lab structure.

Lab data are now available in a `felabs` directory in your home directory. For each lab there is a directory containing various data. This directory can also be used as a working space for each lab so that you properly keep the work on each lab well-separated.

Exit Synaptic if it is still open. If you don't, you won't be able to run `apt-get install` commands, because only one package management tool is allowed at a time.

You are now ready to start the real practical labs!

Install extra packages

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

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

More guidelines

Can be useful throughout any of the labs

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

sources, compiling...) can be done as a regular user.

- If you ran commands from a root shell by mistake, your regular user may no longer be able to handle the corresponding generated files. In this case, use the `chown -R` command to give back the new files to your regular user.
  Example: `chown -R myuser.myuser linux-2.6.25`
- In Debian, Ubuntu and other derivatives, don't be surprised if you cannot run graphical applications as `root`. You could set the `DISPLAY` variable to the same setting as for your regular user, but again, it's unnecessary and unsafe to run graphical applications as `root`.

Sysdev - 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/<user>/felabs/sysdev/toolchain` directory.

Make sure you have at least 2 GB of free disk space.

Install needed packages

Install the packages needed for this lab:

```
sudo apt-get install autoconf automake libtool
sudo apt-get install libncurses-dev bison flex patch
sudo apt-get install cvs texinfo build-essential
sudo apt-get clean
```

Getting Crosstool-ng

Get the latest 1.12.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/overview.txt`, do:

```
./configure --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 the prefix directory to `/usr/local/xtools/${CT_TARGET}`
  This is the place where the toolchain will be installed.
- Change the number of parallel jobs to 2 times the number of CPU cores in your workstation. Building your toolchain will be faster.

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.

In Debug facilities:

- Enable gdb, strace and ltrace. Remove the other options (dmalloc and duma). In gdb options, enable the “Cross gdb” and “Build a static gdbserver” options; the other options are not needed.

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.

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!

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 almost 2 GB of storage space, remove the `.build/` subdirectory in the Crosstool-ng directory.

Bootloader – U-Boot

Objectives: Set up serial communication, compile and install the X-Loader and U-Boot bootloaders, 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, falls into the second category. The monitor integrated in the ROM reads the SD card to search for a valid bootloader before looking at the internal NAND flash for a bootloader. Therefore, by using a SD card, we can start up a OMAP3-based without having anything installed on it.

Setup

Go to the `/home/<user>/felabs/sysdev/u-boot/` directory.

MMC/SD card setup

The ROM monitor can read files from a FAT filesystem on the SD card. However, the SD card must be carefully partitionned, and the filesystem carefully created in order to be recognized by the ROM monitor. Here are the 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. Let's assume that this device is `/dev/sdb`.

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

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 (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 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.

In a terminal open the block device with `fdisk`:

```
$ sudo fdisk /dev/sdb
```

Display the on-line help by pressing the `m` key:

```
Command (m for help): m
Command action
   a   toggle a bootable flag
   b   edit bsd disklabel
   c   toggle the dos compatibility flag
   d   delete a partition
   l   list known partition types
   m   print this menu
   n   add a new partition
   o   create a new empty DOS partition table
   p   print the partition table
   q   quit without saving changes
   s   create a new empty Sun disklabel
   t   change a partition's system id
   u   change display/entry units
   v   verify the partition table
   w   write table to disk and exit
   x   extra functionality (experts only)
```

Print the current partition table typing `p`:

```
Command (m for help): p

Disk /dev/sdb: 1967 MB, 1967128576 bytes
```

Write down the total size.

Let's enter the expert mode for geometry settings:

```
Command (m for help): x
```

We must set the geometry to 255 heads, 63 sectors and calculate the number of cylinders corresponding to your MMC card.

```
Expert command (m for help) : h
Number of heads (1-256, default 4): 255

Expert command (m for help) : s
Number of sectors (1-63, default 62): 63
Warning : setting sector offset for DOS compatibility
```

Now for the number of cylinders, we consider the global size (1967128576 bytes) then divide it by (255*63*512) which gives around 239.16 cylinders. We must round it down to 239.

```
Expert command (m for help) : c
Number of cylinders (1-1048576, default 4): 239
```

After these geometry settings, exit expert mode ('`r`' command) then print the partition table again to check geometry:

```
Command (m for help): p

Disk /dev/sdb: 1967 MB, 1967128576 bytes
```

If any partition exists, delete it ('d' command).

Now, let's create the boot partition:

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
```

```
Partition number (1-4): 1
First cylinder (1-239, default 1): 1
Last cylinder, +cylinders or +size{K,M,G} (1-239, default 239): +64M
```

Mark it bootable:

```
Command (m for help): a
Partition number (1-4): 1
```

Then we change its type to FAT32

```
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): c
Changed system type of partition 1 to c (W95 FAT32 (LBA))
```

Now write your changes and exit:

```
Command (m for help): w
The partition table has been altered!
...
```

Format this new partition:

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

Then, remove and insert your card again.

Your MMC/SD card is ready to use.

X-loader setup

Download the X-loader source code from the IGEP support website:

```
wget http://downloads.igep.es/sources/x-loader-1.4.4-2.tar.gz
tar xvf x-loader-1.4.4-2.tar.gz
cd x-loader-1.4.4-2
```

We need to compile two versions of the bootloader:

- one that will be loaded by processor from the MMC card
- one that we will flash in the internal NAND flash (so that the IGEP can boot without an MMC card).

In order to compile the X-loader, you need to:

- set the `CROSS_COMPILE` environment variable:
  `export CROSS_COMPILE=arm-linux-`
- specify the PATH to the toolchain that you made:
  `export PATH=/usr/local/xtools/arm-unknown-linux-uclibcgnueabi/bin:$PATH`
- run `make igep0020-sdcard_config` to set the configuration for the target board. This one is intended to be loaded by the processor from the MMC card.
- run `make` to build the image

The resulting file is stored in the x-loader main directory as `x-load.bin`. This file must be "signed" in order to be executed by the processor. Using the `signGP` tool in the `contrib/` subdirectory, sign the `x-load.bin` file:

`./contrib/signGP x-load.bin`

This produces an `x-load.bin.ift` file. You can copy it to the MMC

In fact, for a General Purpose (GP) device, the signature only consists of adding a "Table of Contents" at the beginning of the image, explaining which program to execute.

card, renaming it as MLO.

Now, we need the X-loader that will be flashed:

- run make distclean to remove compiled files
- run make igep0020-flash_config
- run make to build it
- sign it the same way as before, using signGP

The Flash chip on the IGEP has a specific feature that may makes it impossible for the first stage bootloader to load data from the flash: the chip has 2 planes of data and 2 consecutive blocks of data won't be on the same plane. But the monitor in ROM will only load data from one plane, so we need to replicate every block of the bootloader twice.

Now, from the X-Loader source directory, run the genddp.sh script that is available in the tools/ subdirectory of the lab directory. This results in a file named x-load-ddp.bin.ift ; copy it to the MMC card.

U-Boot setup

Download U-Boot from the mainline igep download site:

```
wget http://downloads.igep.es/sources/u-boot-arm-2010.06-2.tar.gz
tar xvf u-boot-arm-2010.06-2.tar.gz
cd u-boot-arm-2010.06-2
```

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 (you should have already done it when you compiled X-loader);
- run make <NAME>_config, where <NAME> is the name of a configuration file in include/configs/. For our platform, the configuration file is include/configs/igep0020.h. Read this file to get an idea of how a U-Boot configuration file is written;
- Finally, run make, which should build U-Boot.

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.

You can now copy the generated u-boot.bin file to the MMC card.

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
```

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 now boots from the MMC card:

```
Texas Instruments X-Loader 1.4.4-2 (Mar  7 2011 -
12:24:42)
Reading boot sector
Loading u-boot.bin from mmc

U-Boot 2010.06-2 (May 13 2011 - 12:13:22)

OMAP3630/3730-GP ES2.0, CPU-OPP2, L3-165MHz
IGEP v2 board + LPDDR/ONENAND
I2C:   ready
DRAM:  512 MiB
Muxed OneNAND(DDP) 512MB 1.8V 16-bit (0x58)
OneNAND version = 0x0031
Chip support all block unlock
Chip has 2 plane
block = 2048, wp status = 0x2
Scanning device for bad blocks
OneNAND: 512 MiB
OneNAND: Read environment from 0x00200000
In:    serial
Out:   serial
Err:   serial
Die ID #4d5400011ff0000001592f350202c01d
Net:   smc911x-0
Hit any key to stop autoboot:  3
```

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 first reflash the X-Loader in NAND. To do so, type the following commands:

```
mmc init 0
```

This initializes the MMC interface

```
fatload mmc 0 81000000 x-load-ddp.bin.ift
```

This loads the file from MMC 0 partition 0 to memory at address 0x81000000

```
onenand erase 0 80000
```

This command erases a 0x80000 byte long space of the NAND flash from offset 0

```
onenand write 81000000 0 80000
```

This command writes data to the NAND flash. The source is

0x81000000 (where we've loaded the file to store in the flash) and the destination is offset 0 of the 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.

X-Loader has been transfered to NAND flash. You can now do the same with U-Boot. The storage offset in the NAND is `0x80000` (just after the space reserved for X-Loader) and the length is `0x180000`.

After flashing the U-Boot image, you can remove MMC card, then reset the IGEP board. You should see the freshly flashed X-Loader and U-Boot starting.

You should now see the U-Boot prompt:

```
U-Boot #
```

You can now use U-Boot. Run the `help` command to see the available commands.

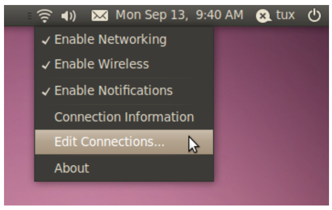
Setting up Ethernet communication

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`. Select the new wired network connection, and in the `IPv4 Settings` tab, 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).



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
```

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
```

In case it is not set yet, you may also need to configure the MAC address for the board:

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

To make these settings permanent, save the environment:

```
saveenv
```

Now switch your board off and on again.

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.

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
```

This should download the file `textfile.txt` 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.

Kernel - 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

Go to the `/home/<user>/felabs/sysdev/kernel` directory.

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-2.6.38.x`, which this lab was tested with.

To practice the `patch` command later, download the full `2.6.37` sources. Unpack the archive, which creates a `linux-2.6.37` directory.

For your convenience, you may copy the source URL from your web browser and then use `wget` to download the sources from the command line:

```
wget <url>
```

`wget` can continue interrupted downloads

Apply patches

Install the patch command, either through the graphical package manager, or using the following command line:

```
sudo apt-get install patch
```

Download the 2 patch files corresponding to the latest `2.6.38` stable release: a first patch to move from `2.6.37` to `2.6.38` and a second patch to move from `2.6.38` to `2.6.38.x`.

Without uncompressing them (!), apply the 2 patches to the `linux-2.6.37` 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?

Did you know it? `gvim` can open compressed files on the fly!

Vim supports syntax highlighting for patch files

Rename the `linux-2.6.37` directory to `linux-2.6.38.<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/<user>/felabs/sysdev/kernel` directory.

Install the following packages:
`libqt4-dev` and `uboot-mkimage`.

`libqt4-dev` is needed for `make xconfig`
`uboot-mkimage` is needed to build the `uImage` 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 install the 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
```

Makefile setup

Modify the toplevel Makefile file to cross-compile for the `arm` platform using the above toolchain.

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!

For later use, we need to edit a bit the configuration. Change the kernel compression from Gzip to LZMA. This compression algorithm is far more efficient than Gzip, in terms of compression ratio, at the expense of an 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.

Look at the end of the kernel build output to see which file contains the kernel image.

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 `uboot-mkimage` package. To run the post-processing operation on the kernel image, simply run:

`make uImage`.

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, make sure your `uImage` file is in the directory exposed by the TFP server.
- On the target, load `uImage` from TFTP into RAM at address `0x80000000`:
  `tftp 0x80000000 uImage`
- Boot the kernel:
  `bootm 0x80000000`.

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

```
setenv bootcmd 'tftp 80000000 uImage;bootm 80000000'
saveenv
```

Flashing the kernel in NAND flash

In order to let the kernel boot on the board autonomously, we can flash it in the NAND flash available on the IGEP. The NAND flash can be manipulated in U-Boot using the `onenand` command, which features several subcommands. Run `help onenand` to see the available subcommands and their options.

The NAND flash is logically split in three partitions by the Linux kernel, as defined in the `igep2_onenand_partitions` definition in the board-specific file `arch/arm/mach-omap2/board-igep0020.c`. The first 3 partitions are dedicated to X-loader, to U-boot and to its environment variables. The 4th partition, which is 3 MiB big, from NAND address 0x280000 to 0x580000, is reserved for the kernel.

The easiest way to compute these start and end addresses is to read them in the kernel bootup messages!

So, let's start by erasing the corresponding 3MiB of NAND flash:

```
onenand erase 0x280000 0x300000
```
(NAND addr) (size)

Then, copy the kernel from the MMC/SD card into memory, using the same address as before.

Then, flash the kernel image:

```
onenand write 0x80000000 0x280000 0x300000
```
(RAM addr) (NAND addr) (size)

Then, we should be able to boot the kernel from the NAND using:

```
onenand read 0x80000000  0x280000    0x300000
```
(RAM addr) (NAND addr) (size)

```
bootm 0x80000000
```

`onenand read` copies the kernel to RAM and then, `bootm` executes it.

Write an U-Boot script that automates the kernel download and flashing procedure. Finally, adjust the `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.

Sysdev – Tiny embedded system with BusyBox

Objective: making a tiny yet full featured embedded system.

After this lab, you will

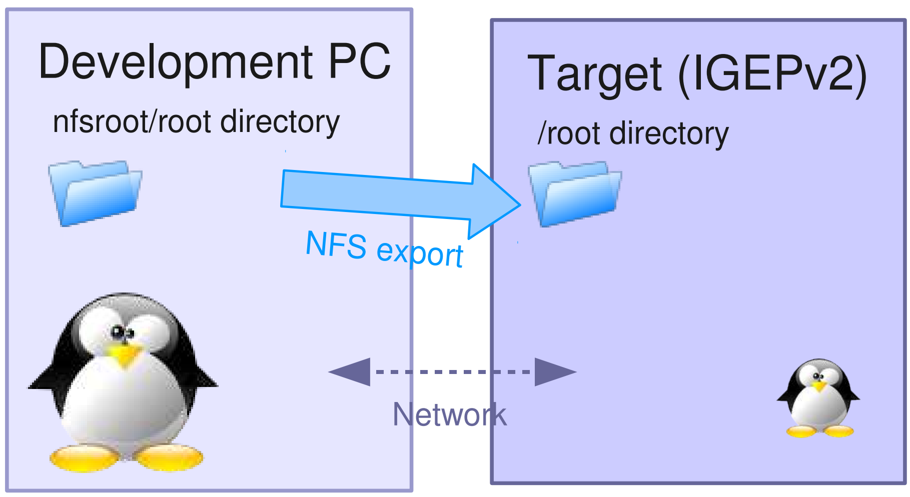
- be able to configure and build a Linux kernel that boots on a directory on your workstation, shared through the network by NFS.
- be able to create and configure a minimalistic root filesystem **from scratch** (ex nihilo, out of nothing, entirely hand made...) for the IGEP board
- understand how small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

Get the sources from the latest 2.6.38.x release and place them in the current directory. You can reuse sources from previous labs.

Kernel configuration

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.

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, create 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 (the path and the options must be on the same line!)

```
/home/<user>/felabs/sysdev/tinysystem/nfsroot
192.168.0.100(rw,no_root_squash,no_subtree_check)
```

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):

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

Caution: type ttyO2 (“O” like “OMAP”), not zero (tty02)

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

Booting the system

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

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, the kernel will complain 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 empty, there isn't such an application. Follow on to add Busybox to our root filesystem and finally make it useful.

Root filesystem with Busybox

Download the latest BusyBox 1.19.x release and configure it with the configuration file provided in the `data/` directory.

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.

At least, make sure you build BusyBox statically!

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

Install BusyBox in the root filesystem by running `make install`.

Try to boot your new system on the board. If everything goes right, the kernel should again confirm that it managed to mount the NFS root filesystem. Then, you should get errors about missing `/dev/ttyX` files. Create them with the `mknod` command (using the same major and minor number as in your GNU/Linux workstation). Try again.

You can understand our approach to build filesystems from scratch. We're waiting for programs to complain before adding device or configuration files. This is a way of making sure that every file in the filesystem is used.

At the end, you will access a console and will be able to issue commands through the default shell.

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.

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.

Actually, you will probably have several instructive surprises when trying to implement this. Don't hesitate to share your questions with your instructor!

Any issue after doing this?

Switching to shared libraries

Take the `hello.c` program supplied in the `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, recompile Busybox without the static compilation option, so that Busybox takes advantages of the shared libraries that are now present on the target.

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 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?

How much RAM does your system need?

Check the `/proc/meminfo` file and see how much RAM is used by your system.

The total amount of RAM shown by `/proc/meminfo` is the total physical RAM minus the space taken by the kernel code and its static data.

You can try to boot your system with less memory, and see whether it still works properly or not. For example, to test whether 20 MB are enough, boot the kernel with the `mem=20M` parameter. Linux will then use just 20 MB of RAM, and ignore the rest.

Try to use even less RAM, and see what happens.

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

Go to `/home/<user>/felabs/sysdev/fs`.

Reuse the kernel that you used in `/home/<user>/felabs/sysdev/tinysystem`.

Recompile it with support for `SquashFS` and `ext3`.

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

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

Add partitions to the MMC card

Using fdisk, add two additional partitions to the MMC card (in addition to the existing “boot” partition created in the bootloaders lab) :

- One partition, of at least 1 MB, that will be used for the root filesystem. Due to the geometry of the device, the partition might be larger than 1 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.

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 (probably /dev/sda1), and the device that your workstation uses (probably /dev/sdb1).

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 while it's running Linux. Using the dmesg command, or having a look at the console, see how the kernel detects the partitions and which device names it gives to them.

Modify the setup scripts in your root filesystem to mount the third disk partition on /www/upload/files.

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.

Modify the www/cgi-bin/upload.cfg configuration file to store the log file in /var/log/upload.log. You will loose 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.

Before changing your startup scripts, you may also try your mount command in the running system, to make sure that it works as expected.

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.

If you don't do this, you will get a kernel panic, because of a failure to mount the root filesystem, being not ready yet.

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

Store the kernel image on the MMC card

Finally, copy the `uImage` kernel image 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 from the MMC card instead of loading the kernel through the network.

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/<user>/felabs/sysdev/fs`.

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 OneNAND flash will be :

- From 0 to 0x80000, X-Loader (512 KB)
- From 0x80000 to 0x200000, U-Boot (1536 KB)
- From 0x200000 to 0x280000, U-Boot environment (512 KB)
- From 0x280000 to 0x680000, Linux kernel (4 MB)
- From 0x680000 to 0x880000, the JFFS2 root filesystem (2 MB) as read-only
- From 0x880000 to the end, the JFFS2 data filesystem

Filesystem image preparation

Prepare a JFFS2 filesystem image from the `/www/uploads/files` directory from the previous lab, specifying an erase block size of 256 KiB and a page size of 4 KiB. Enable the "pad" option, don't add cleanmarkers.

Modify the `/etc/init.d/rcS file` to mount a JFFS2 filesystem on the sixth flash partition, instead of an ext3 filesystem on the third MMC disk partition. Create a JFFS2 image for your root filesystem. with the same options as the data filesystem.

Enabling NAND flash and filesystems

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_ONENAND_OMAP2`).

Because checking subpages writes of 2KiB-pages NANDs isn't working yet, you will need to make sure that `CONFIG_MTD_ONENAND_VERIFY_WRITE` isn't enabled.

In the rest of the lab, we also assume that `CONFIG_MTD_ONENAND_2X_PROGRAM` is enabled.

Update your kernel image on flash.

MTD partitioning and flashing

Memory layout and partitioning can be defined inside kernel sources, naturally in the `arch/<arch>/<march>/<board>.c` since it is board dependent. Nevertheless, during device development, it can be useful to define partitions at boot time, on the kernel command line.

Enter the U-Boot shell and erase the NAND flash, from offset 0x00280000, up to the end of the NAND flash (Erase size : 0x1FD80000 bytes)

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

Using the `tftp` command, download and flash the JFFS2-ro image the correct location.

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

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

Look at the way MTD partitions are defined in the kernel sources (arch/arm/mach-omap2/board-igep-0020.c)

Set the `bootargs` variable so that :

- you define the 6 MTD partitions, as detailed previously
- the root filesystem is mounted from the 5th partition, and is mounted read-only (kernel parameter `ro`)

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.

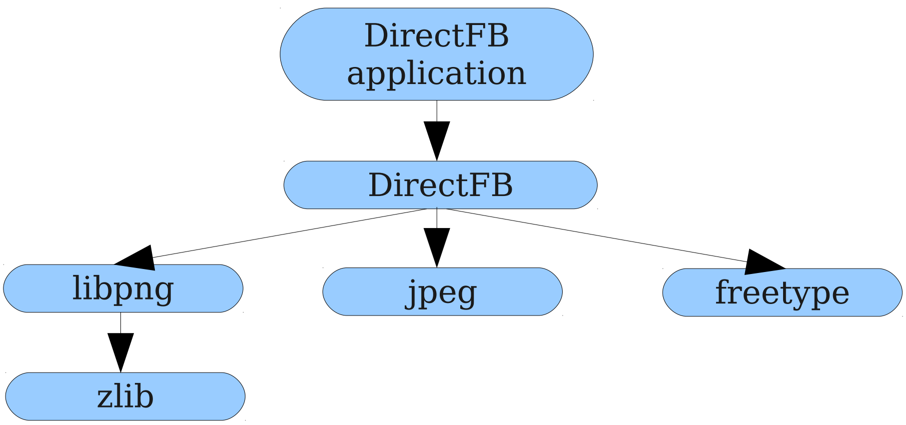
We'll see that manually re-using existing libraries is quite tedious, so that more automated procedures are necessary to make it easier. However, learning how to perform these operations manually will significantly help you when you'll face issues with more automated tools.

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/<user>/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/<user>/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.

Modify the `/etc/qemu-ifup` script so that it just contains 2 lines:

```
#!/bin/sh
/sbin/ifconfig $1 172.20.0.1
```

By default, Qemu configures bridged networking, but we will use a routed network instead.

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/<user>/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.1.2.3 libz.so
ln -s libz.so.1.2.3 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/<user>/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/<user>/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/arm-
unknown-linux-uclibcgnueabi/sys-
root/usr/bin/../lib/gcc/arm-linux-
uclibcgnueabi/4.3.3/../../../../arm-linux-
uclibcgnueabi/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/<user>/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/<user>/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<include dir> if you have headers
                  in a nonstandard directory <include dir>
```

Let's use it:

```
 LDFLAGS=-
L/home/<user>/felabs/sysdev/thirdparty/staging/usr/lib \
 CPPFLAGS=-
I/home/<user>/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 informations 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/<user>/felabs/sysdev/thirdparty/staging/ install
```

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

```
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, thanks to the patch applied previously:

```
make
DESTDIR=/home/<user>/felabs/sysdev/thirdparty/staging/ install
```

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

```
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/<user>/felabs/sysdev/thirdparty/staging/ install
```

And install only the required files in the «target» space:

```
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 -O3 -ffast-math -pipe
-D_REENTRANT   conftest.c -ljpeg  -ldl -lpthread  >&5
/usr/local/xtools/arm-unknown-linux-uclibcgnueabi/arm-unknown-
linux-uclibcgnueabi/sys-root/usr/bin/../lib/gcc/arm-linux-
uclibcgnueabi/4.3.3/../../../../arm-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/<user>/felabs/sysdev/thirdparty/staging/usr/lib \
 CPPFLAGS=-
I/home/<user>/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/<user>/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.

Unfortunately, This is only possible with `pkg-config` >= 0.23, which is not yet available in the Ubuntu distribution. So start by installing the `pkg-config` Ubuntu package available in the `data/` directory of the lab:

```
sudo dpkg -i data/pkg-config_0.23-1_i386.deb
```

Then, let's run the configuration of DirectFB again, passing the `PKG_CONFIG_PATH` and `PKG_CONFIG_SYSROOT_DIR` environment variables:

```
LDFLAGS=-L/home/<user>/felabs/sysdev/thirdparty/staging/usr/lib \
 CPPFLAGS=-
I/home/<user>/felabs/sysdev/thirdparty/staging/usr/include \
 PKG_CONFIG_PATH=/home/<user>/felabs/sysdev/thirdparty/staging/us
r/lib/pkgconfig/ \
 PKG_CONFIG_SYSROOT_DIR=/home/<user>/felabs/sysdev/thirdparty/sta
ging/ \
 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 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/<user>/felabs/sysdev/thirdparty/staging/ install
```

And so the installation in the «target» space:

- First, the libraries:
  `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/<user>/felabs/sysdev/thirdparty/staging/usr/lib \
 CPPFLAGS=-
I/home/<user>/felabs/sysdev/thirdparty/staging/usr/include \
 PKG_CONFIG_PATH=/home/<user>/felabs/sysdev/thirdparty/staging/us
r/lib/pkgconfig/ \
 PKG_CONFIG_SYSROOT_DIR=/home/<user>/felabs/sysdev/thirdparty/sta
ging/ \
 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/<user>/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:

```
	cp -a /usr/local/xtools/arm-unknown-linux-
uclibcgnueabi/arm-unknown-linux-uclibcgnueabi/sys-
root/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 /usr/local/xtools/arm-unknown-linux-
uclibcgnueabi/arm-unknown-linux-uclibcgnueabi/sys-
root/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 /usr/local/xtools/arm-unknown-linux-
uclibcgnueabi/arm-unknown-linux-uclibcgnueabi/sys-
root/lib/libgcc_s* target/lib
```

Now, the application should no longer complain about missing library. But when started, should complain about inexisting `/dev/fb0` device file. So let's create this device file:

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

Next execution of the application will complain about missing `/dev/tty0` device file, so let's create it:

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

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

```
	Unable to dlopen '/usr/lib/directfb-1.4-
0/interfaces/IDirectFBImageProvider/libidirectfbimageprovi
der_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 /usr/local/xtools/arm-unknown-linux-uclibcgnueabi/arm-unknown-linux-uclibcgnueabi/sys-root/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 into the `/home/<user>/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://www.buildroot.net. Download the stable 2011.08 version which we have tested for this lab. Uncompress the tarball and go inside the `buildroot` 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;
- `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;
- `target` contains patches and other items specific to particular hardware platforms
- `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
- Target Architecture Variant: `arm926t`.
- Target ABI: `EABI`
- Build options
  - Number of jobs to run simultaneously: choose `2` or `4`, for example, to speed up compiling, especially if you have a dual-core system.
- Toolchain
  - *Toolchain type*: `External toolchain`
  - *External toolchain C library*: `uClibc`
  - We must tell Buildroot about our toolchain configuration, so: enable Large File Support, RPC. Buildroot will check these parameters anyway.
  - *External toolchain path*: use the toolchain you built: `/usr/local/xtools/arm-unknown-linux-uclibcgnueabi`
- System configuration
  - Port to run a getty (login prompt) on, change `ttyS0` to `tty1`
- Package selection for the target
  - Keep Busybox (default version) and keep the Busybox configuration proposed by Buildroot;
  - In Graphics libraries and applications
    - Select DirectFB. Buildroot will automatically select the necessary dependencies.
    - Remove touchscreen support from DirectFB
    - Select DirectFB examples
    - Select all the DirectFB examples
- Target filesystem options
  - Select `ext2 root filesystem`
  - Select `tar root filesystem`

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

Generate the embedded Linux system

Just run

```
make
```

It fails quickly because we lack the `gettext` and `subversion` packages, so install them. 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/` directory. 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 don'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 most 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 an 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.

The `run_qemu` script contains what's needed to boot the system in Qemu.

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

```
df_andi
```

```
df_dok
df_fire
...
```

Tests on the IGEPv2 board

If you have graphical displays to connect your IGEPv2 board to (LCD display with DVI-D or HDMI input, TI Pico Projector...), and the corresponding cables (HDMI to DVI-D or HDMI to HDMI), you can test your root filesystem on the board.

You can skip to the next section if you don't have a display.

Here's what you can do:

- Edit `etc/inittab`, uncomment the line with `ttyS0` and replace all the occurrences of `ttyS0` by `ttyO2`
- Either copy the ext2 image to a block device that can be accessed by your hardware (flash card reader, USB drive...)
- Or mount the ext2 image and copy its contents to a flash partition on your board.
- Connect your board to an DVI-D or HDMI display
- Get the latest Linux 2.6.39.x sources, take the default configuration of OMAP2 and build this Linux kernel with the following settings:
  `CONFIG_OMAP2_DSS=y`
  `FB_OMAP2=y`
  `CONFIG_PANEL_GENERIC_DPI=y`
  `LOGO=y` (optional)
- Add the following settings to the boot arguments:
  - `console=tty0`
    (allows you to have both a framebuffer and serial console)
  - `vram=12M` (video RAM)
  - `omapfb.mode=dvi:640x480MR-16@60` (example for the Pico Projector. You may first try to do without this parameter, and then specify a mode that your monitor supports).
  - `omapdss.def_disp=dvi` (default output for the Omap Display SubSystem driver)

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.

If you haven't done it yet in the previous lab, you will also need to modify the `/etc/qemu-ifup` script to contain:

```
#!/bin/sh
/sbin/ifconfig $1 172.20.0.1
```

Sysdev - Application development

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

Setup

Go to the `/home/<user>/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, adapt the `run_qemu` script to your configuration, and verify 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/<user>/felabs/sysdev/buildroot/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:

```
export
PKG_CONFIG_PATH=/home/<user>/felabs/sysdev/buildroot/outpu
t/staging/usr/lib/pkgconfig
```

```
export
PKG_CONFIG_SYSROOT_DIR=/home/<user>/felabs/sysdev/buildroo
t/output/staging/
```

```
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!

Sysdev - Remote application debugging

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

Setup

Go to the `/home/<user>/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, strace and ltrace

`gdbserver`, `strace` and `ltrace` 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.

A work around 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` (implying `/dev/console`)
by
`ttyO2::askfirst:/bin/sh` (using `/dev/ttyO2`)

Also create `/dev/ttyO2` (major `4`, minor `66`, like `ttyS2`) and 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, 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 ltrace

Try to run the program through `ltrace`. You will see that another library is required to run this utility. Find this library in the toolchain and add it to your root filesystem again.

Now, `ltrace` should run fine and you should see what the program does: it tries to consume as much system memory as it can!

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. Keep this binary on your workstation, and make a copy in the `/root` directory of the target root filesystem. Then, strip the binary on the target to remove the debugging symbols. They are only needed on your host, where the cross-debugger will run:

```
arm-linux-strip path/to/root/filesystem/root/vista-emulator
```

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:

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

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` and `ltrace`.
- 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 apply the rt-preempt patches and be more familiar with kernel configuration settings related to real-time.
- Be able to check clock accuracy.
- Be able to start processes with real-time priority.
- Have compared scheduling latency on your system, between a standard kernel and a kernel with real-time preempt patches.

Setup

Go to the `/home/<user>/felabs/`**`realtime`**`/rttest` directory.

For this lab, you must have compiled a 2.6.35.9 kernel for the IGEP board, with the default configuration (named `igep0020_defconfig`, the generic `omap2plus_defconfig` didn't exist in 2.6.35.9), except that you will have to remove the libertas wireless modules. Remove the `CONFIG_HIGH_RES_TIMERS` option, to first test the kernel without high-resolution timers.

You will have to log with username `root` and no password.

Boot the IGEP board by mounting the root filesystem available at `/home/<user>/felabs/`**`realtime`**`/rttest/nfsroot/` with NFS. Note that in 2.6.35.9, the OMAP serial port were named `ttyS`, not `ttyO`, so you must adjust your `console=` argument to use `ttyS2`.

Please stay with a 2.6.35.9 version, as this is the most recent version with Xenomai support.

Install `netcat` on your host, by running:

```
apt-get install netcat
```

Download CodeSourcery's `2009q1` toolchain at:

http://www.codesourcery.com/sgpp/lite/arm/portal/release858

Choose "IA32 GNU/Linux TAR"

Untar it.

Add `/home/<user>/felabs/`**`realtime`**`/rttest/arm-2009q1/bin` to your `PATH`.

We are using a glibc toolchain because glibc has better support for the POSIX RT API than uClibc. In our case, when we created this lab, uClibc didn't support the `clock_nanosleep` function used in our `rttest.c` program.

Using high-resolution timers

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 following options in the kernel configuration:

- `CONFIG_HIGH_RES_TIMERS`

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 `doload 300 > /dev/null 2>&1 &` on the board, and using `netcat 192.168.0.100 5566` on your workstation when you see the message “`Listening on any address 5566`” 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).

Re-do the simple tests with this new preemptible kernel and compare the results.

Testing Xenomai scheduling latency

Get Xenomai from its download area:

http://download.gna.org/xenomai/stable/

Untar Xenomai.

Prepare the kernel for Xenomai compilation :

```
./scripts/prepare-kernel.sh \
      --arch=arm --linux=/path/to/linux-2.6.35.9
```

You can reuse the kernel configuration from a previous compile job, then launch kernel configuration tool again and enable the options:

- `CONFIG_XENOMAI`
- `CONFIG_XENO_DRIVERS_TIMERBENCH`

Other options of interest (ARM specific) are:

- `CONFIG_XENO_HW_UNLOCKED_SWITCH`

Read the help associated with these options, decide whether you want to enable them.

Compile `rttest` for the Xenomai POSIX skin:

```
DESTDIR=/home/<user>/felabs/realtime/rttest/nfsroot/
export DESTDIR
```

```
CFL=`$DESTDIR/usr/bin/xeno-config --skin=posix --cflags`
LDF=`$DESTDIR/usr/bin/xeno-config --skin=posix --ldflags`
arm-none-linux-gnueabi-gcc $CFL -o rttest rttest.c $LDF
```

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.

Re-run the tests, compare the results.

Testing Xenomai interrupt latency

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

```
latency -t 2
```

Using mdev

Objective: Practicing with BusyBox mdev

After this lab, you will be able to

- Use `mdev` to populate the `/dev` directory with device files corresponding to devices present on the system.
- Use `mdev` to automount external disk partitions.

Root filesystem

We will reuse the root filesystem from the “Tiny system” lab.

Kernel settings

Reuse the Linux kernel from the “Tiny system” lab. If you prefer to start from fresh sources, use the configuration supplied in the `data` directory.

Now add or modify the below settings to your kernel:

- Enable loadable module support: CONFIG_MODULES=y
- Module unloading: CONFIG_MODULE_UNLOAD=y
- Support for Host-side USB: CONFIG_USB=**m**
  Make sure this is set as a module!
- OHCI HCD support: CONFIG_USB_OHCI_HCD=**m**
- USB Mass Storage support: CONFIG_USB_STORAGE=**m**
- And any other feature that could be needed on your hardware to access your hot-pluggable device.

Compile your kernel. Install the modules in your root filesystem using:

```
make INSTALL_MOD_PATH=<root-dir-path> modules_install
```

Booting the system

Boot your system through NFS with the given root filesystem.

To make sure that module loading works, try to load the `usb-storage` module:

```
modprobe usb-storage
```

First mdev tests

We are first going to use `mdev` to populate the `/dev` directory with all devices present at boot time.

Modify the `/etc/init.d/rcS` script to mount a tmpfs filesystem on `/dev/`, and run `mdev -s` to populate this directory with all minimum device files. Very nice, isn't it?

Using mdev as a hotplug agent

Using the guidelines in the lectures, and BusyBox documentation, use `mdev` to automatically create all the `/dev/sd[a-z][1-9]*` device files when a USB disk is inserted, corresponding to the disk itself and its partitions.

Also make sure these device files are also removed automatically when the flash drive is removed.

Automatic mounting

Refine your configuration to also mount each partition automatically when a USB disk is inserted, and to do the opposite after the disk is removed.

You could use `/media/<devname>` as mount point for each partition.

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.lzma` archive that you can copy to a USB flash drive, for example. This file should only be a few hundreds of MB big.