Embedded Linux Training

Lab Book

eMagii
http://www.emagii.com

Free Electrons
http://free-electrons.com

December 5, 2014

About this document

Updates to this document can be found on http://www.emagii.com/doc/training/sysdev/.

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

More details about our training sessions can be found on http://www.emagii.com/training.

Copying this document

Setting up Ubuntu

General Setup of Ubuntu

Install Synaptic

```
$ sudo apt-get install synaptic
$ sudo su
$ synaptic&
```

Start `synaptic` and install

- `gnome`
- `nautilus-open-terminal`
- `git`
- `git-core`
- `samba`
- `system-config-samba`

Log out

Click on the white `Ubuntu` button when you get the login screen and select the `Gnome Classic` option. Log again.

`Gnome Classic` will from now be your default.

Open a terminal

Since you installed `nautilus-open-terminal`, you can open a terminal by right clicking the mouse, and select the terminal.

Make sure bash is the default shell

The normal Ubuntu installation uses the `dash` shell which won't work.

Changed to the `bash` shell.

```
$ cd /bin/
$ ls -l sh
lrwxrwxrwx 1 root root 9 dec  6 15:25 sh -> /bin/dash
$ sudo unlink sh
$ sudo ln -s /bin/bash sh
$ ls -l sh
lrwxrwxrwx 1 root root 9 dec  6 15:25 sh -> /bin/bash
```


full-sysdev-labs.pdf

Generate ssh keys

If you already have `rsa` keys in the `$HOME/.ssh` directory, you can skip this step.

If not, you generate the keys like this (Make sure you are not running as super-user)

```
$ ssh-keygen -t rsa
```

Use the default location and provide a password (twice).

This will generate

- Private Key: `$HOME/.ssh/id_rsa`
- Public Key: `$HOME/.ssh/id_rsa.pub`

The **Private Key** should **never** be give out to anyone else.

Update Ubuntu to the latest package versions

Caution: Do not `upgrade` to `Ubuntu 14.04`

Run the update manager, to update the machine.

This will take some time.

```
Program->System Tools->Administration->Update
```

Make sure pkg-config works

Edit `${HOME}/.bashrc` and add

```
export PKG_CONFIG_PATH=/usr/lib/x86_64-linux-gnu/pkgconfig
```

You probably have to restart the computer afterwards.

Setting up the git client

Objective: Get a working git

After this lab, you will be able to work with the **git** source code control system

Install the git client

If you do not have git installed, you need to do this now.

```
sudo apt-get install git git-core
```

You should now set up your git personal information

Use the example below, but with your own name/email:

```
git config --global user.name "Allan K Luring"
git config --global user.email "allan@luring.com"
```

You can get help on git, using the `git help` command.

Setting up the Training

Download files and directories used in practical labs

Install lab data

For the different labs in the training, your instructor has prepared a set of data (kernel images, kernel configurations, root filesystems and more). Clone the lab directory to your home directory.

```
cd
git clone https://github.com/emagii/Training-Labs.git felabs
```

If you are using Ubuntu 14.04, you need to checkout the right version

```
cd felabs
git checkout -b 14.04 origin/14.04
```

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

Install extra packages

You will need to have a number of packages installed on your machine.

Go to the `~/felabs/sysdev` directory and install required packages using the Makefile.

```
make prepare
```

Since the install requires `root` privilegues, you will have to supply the `root` password.

Configure Your lab network

Edit the `host.mk` and change the **SERVER_IP** and **IPADDR** variables if they conflict with your main network during the lab.

Check with `ifconfig` if you do not know which network you are using.

More guidelines

Can be useful throughout any of the labs

- Read instructions and tips carefully. Lots of people make mistakes or waste time because they missed an explanation or a guideline.
- Always read error messages carefully, in particular the first one which is issued. Some people stumble on very simple errors just because they specified a wrong file path and didn't pay enough attention to the corresponding error message.
- Never stay stuck with a strange problem more than 5 minutes. Show your problem to your colleagues or to the instructor.
- You should only use the `root` user for operations that require super-user privileges, such as: mounting a file system, loading a kernel module, changing file ownership, configuring the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.
- If you ran commands from a root shell by mistake, your regular user may no longer be able to handle the corresponding generated files. In this case, use the `chown -R` command to give the new files back to your regular user.
  Example: `chown -R myuser:myuser linux-3.4`

Setting up Applications Switcher

How to get multiple workspaces in Ubuntu 12.04

Use **`Gnome Classic (no effects)`** when you log in.

With effects:

Install Compiz Config Settings Manager:

To open terminal hit Alt+Ctrl+T and run following commands:

```
sudo apt-get install compizconfig-settings-manager
```

Then go to system tools ¿ preferences ¿ compizconfig

Go at the very bottom where it says windows management

Put a checkmark in application switcher and exit.

Then you need to setup the `wmctrl` CLI for talking to X.

```
sudo apt-get install wmctrl
```

Click on the cogwheel icon in the upper right corner and select startup programs

Add `wmctrl -n 1`

To be continued...

Setting up the serial communication

Objective: Get a working serial communication for the console

After this lab, you will be able to communicate with your **Beaglebone Black** over the console

Install the picocom program

If you did not run `make prepare` in `~/felabs/sysdev` , you need to install `picocom` now.

The code below will install `picocom` and make your user belong to the `dialout` group, which is needed for you to be allowed to write to the serial console:

Alternatively:

```
sudo apt-get install picocom
sudo adduser ${USER} dialout
```

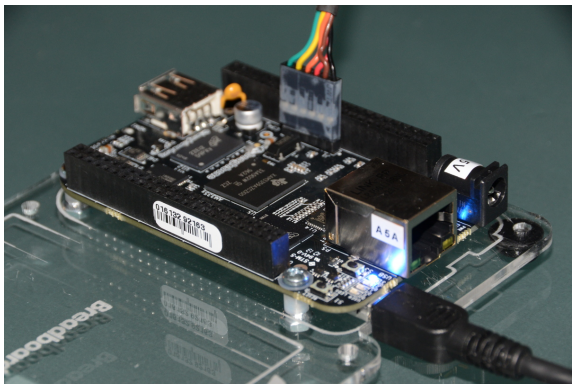
You need to log out and in again for the group change to be effective.

Setting up serial communication with the board

Make sure that the USB-to-Serial cable to the **Beaglebone Black** is disconnected from your computer. Check that no other USB serial ports are connected to the system

```
ls /dev/ttyU*
```

Plug the **Beaglebone Black** into your computer using the provided USB-to-serial cable. When plugged-in, a serial port should appear as `/dev/ttyUSB0` if there are no other USB - Serial ports. Otherwise find out which serial port was just activated using `dmesg`



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

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]`.

Setting up an SD-card

Objectives: Set up an SD-card for use in later labs

MMC/SD card setup

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

First, clean out your system log buffer by `sudo dmesg -c`.

Then list all disk devices by `ls /dev/sd*`.

Connect your card reader to your workstation, with the MMC/SD card inside and again list all the disk devices by `ls /dev/sd*`.

If you see a new disk device appearing, that will be the new SD-card. If you see several new devices like `ls /dev/sde /dev/sde1 /dev/sde2`, this is because the SD-card has several partitions.

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

If you still fail to detect the disk, then type the `dmesg` command to see which device is used by your workstation. In case the device is `/dev/sde`, you will see something like:

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

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

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

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

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

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

```
$ sudo umount /dev/sde1
$ sudo umount /dev/sde2
...
```

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

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

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

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

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

In the `cfdisk` interface create three primary partitions, starting from the beginning:

- BOOT: 512 MB size, a `Bootable` type and a `0C` type (`W95 FAT32 (LBA)`)
- ROOTFS: 2048 MB size and a `83` type (`Linux`)
- DATA: 512 MB size and a `83` type (`Linux`)

Press `Write` when you are done.

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

Format the first partition to FAT32, with the `boot` label (name):

```
sudo mkfs.vfat -n BOOT -F 32 /dev/sde1
```

Then format the second partition to `EXT4`.

```
sudo mkfs -t ext4 -L rootfs /dev/sde2
```

Format the third and last partition to `EXT4`.

```
sudo mkfs -t ext4 -L data /dev/sde3
```

Then, remove and insert your card again.

Your MMC/SD card is ready for use.

Setting up a TFTP Server

Objectives: Set up TFTP communication with the development workstation.

Install the TFTPD server

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.

Normally, if you did `make prepare` before, you should have the TFTP server running on your system. Otherwise do:

```
sudo apt-get install tftpd tftp xinetd
```

Configuring the TFTPD server

The configuration file for `tftpd` is `/etc/xinetd.d/tftp`

An example `tftp` file is in `~/felabs/sysdev` /network

```
service tftp
{
        protocol        = udp
        port            = 69
        socket_type     = dgram
        wait            = yes
        user            = nobody
        server          = /usr/sbin/in.tftpd
        server_args     = /tftpboot
        disable         = no
}
```

Create the configuration file.

The default TFTP directory is `/tftpboot`

if it doesnt exist, create it and make it owned by 'nobody' and make sure that `tftpboot` is easily accessible

```
sudo    mkdir -p        /tftpboot
sudo    chown -R nobody /tftpboot
sudo    chmod -R 777    /tftpboot
```

Restart tftpd by:

```
sudo service xinetd restart
```


Testing the TFTPD server

Create a file in `tftpboot`:

```
cd /tftpboot
echo 111 > testfile.txt
```

Start tftp and get the file

```
cd
tftp localhost

tftp> get testfile.txt
Sent 5 bytes in 0.0 seconds

tftp> quit

cat testfile.txt
```

Setting up a Startech USB - Ethernet adapter

Objective: Get a dedicated network port for communication with the **Beaglebone Black**

After this lab, you have prepared for a second network port allowing you to nfs-mount the **Beaglebone Black** root filesystem without changing the network settings of the primary network port.

Prerequisites

You need a **Startech USB31000SW USB Ethernet** adapter to use the port, but the driver can be installed without the Adapter.

It is assumed, that your system is setup to build kernel modules.

This means that kernel source build directory must be present

```
ls /lib/modules/`uname -r`/build
```

Preparing to build kernel drivers

If you did not run `make prepare` in `~/felabs/sysdev` , you need to do this now.

Build and install the kernel driver

Go to the `~/felabs/sysdev` directory

The `network/startech` directory contains a tarball with the driver.

The tarball has been downloaded from the Startech website, and will be slightly modified using a patch to allow it to build.

From the `~/felabs/sysdev` directory, You can compile and install the driver by:

```
sudo make -C network startech-usb
```

Build and install the kernel driver outside the lab

Extract the tarball, and enter the source directory.

Apply the patches from the `../patches` directory

```
make
sudo make install
```

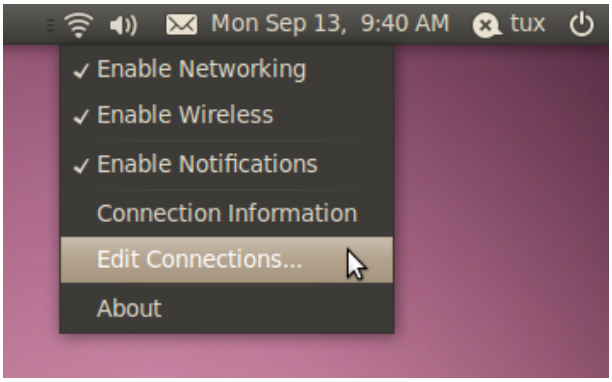

Setting up Ethernet communication

Objectives: Set up a local network connection between your computer and the **Beaglebone Black** .

Connect the host to the Beaglebone Black Board

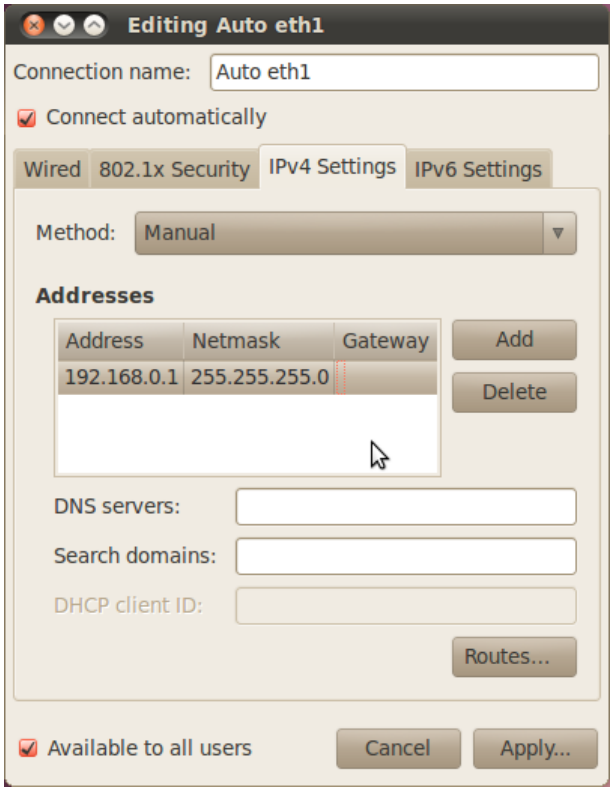
With a network cable, connect the Ethernet port of your board to the one of your computer. If your computer already has a wired connection to the network, your instructor will provide you with a USB Ethernet adapter. A new network interface, probably `eth1` or `eth2`, should appear on your Linux system.

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*:

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



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

The board must also be configured, but this will be done later in the U-Boot lab.

The network port will be present, but its IP address will only be allocated when you have a physical connection from the port to the **Beaglebone Black** .

When you have a running **Beaglebone Black** with a cable inbetween later, you can check by:

```
ifconfig
```


Extra Lab: Setting up a DHCP server

Objective: Use a DHCP server on your USB Network port

After this lab, you will be able to communicate with your **Beaglebone Black** over the console

Install the DHCP daemon

```
sudo apt-get install isc-dhcp-server
```

Configuring the DHCP server daemon

We need to setup

- /etc/default/isc-dhcp-server

```
#Defaults for dhcp initscript
#sourced by /etc/init.d/dhcp
#installed at /etc/default/isc-dhcp-server by the maintainer scripts
#
#This is a POSIX shell fragment
#
#On what interfaces should the DHCP server (dhcpd) serve DHCP requests"
#Separate multiple interfaces with spaces, e.g. eth0 eth1".
INTERFACES="eth2"
```

The only relevant thing is the `INTERFACES` variable, which should be set to the same port as you use for communication with the **Beaglebone Black**

You also need to setup

- /etc/dhcp/dhcpd.conf

```
#
#Sample configuration file for ISC dhcpd for Debian
#
#Attention: If /etc/ltsp/dhcpd.conf exists, that will be used as
#configuration file instead of this file.
#
#
....
option domain-name example.org;
option domain-name-servers ns1.example.org, ns2.example.org;
option domain-name emagii.com;
default-lease-time 600;
max-lease-time 7200;
log-facility local7;
subnet 192.168.0.0 netmask 255.255.255.0 {
range 192.168.0.10 192.168.0.100;
option routers 192.168.0.1;
option subnet-mask 255.255.255.0;
option broadcast-address 192.168.0.254;
option domain-name-servers 192.168.0.1;
option ntp-servers 192.168.0.1;
option netbios-name-servers 192.168.0.1;
option netbios-node-type 8;
 ......
}
```

The interesting things here are

- subnet: Any range which does not collide with your normal network
- server adress: We used the fix IP address of the USB Network adapter.
- range: A range of addresses inside the subnet, but not including the server.

Activating the DHCP daemon

```
sudo service isc-dhcp-server restart
```

Check that it is running:

```
sudo netstat -uap
```

Look for `dhcpd`

Setting up an NFS server

Objective: Prepare the host for NFS booting

Install the NFS Server

`make prepare` should have installed the NFS server. If you do not have an NFS server installed, install it by

```
sudo apt-get install nfs-kernel-server
```

Create the root file system directory

Create a `rootfs` directory in your $HOME directory. This directory will later be used to store the contents of our new root filesystem.

```
mkdir -p /tftpboot/rootfs
chmod 777 /tftpboot/rootfs
```

Configure the NFS server

The NFS configuration file (`/etc/exports`) needs to be modified. Edit the file as `root` and add the following line: (Replace `/home/ulf` with your own home directory)

```
/tftpboot/rootfs *(rw,sync,no_root_squash,no_subtree_check)
```

The format for the line is

```
<directory> <ip-adress>(<options>)
```

The IP address '*' means that the NFS server will allow any computer to connect. You can replace it with the IP address of the **Beaglebone Black** .

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

Then, restart the NFS server:

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

alternatively:

```
sudo exportfs -av
```

You can test your NFS setup by:

```
cd
mkdir nfstest
sudo mount -t nfs localhost:/tftpboot/rootfs nfstest
```


If this appears to hang, then try replacing 'localhost' with your IP number.

```
cd
mkdir nfstest
sudo mount -t nfs <xx.xx.xx.xx>:/tftpboot/rootfs nfstest
```


Extra Lab: Building a cross-compiling toolchain using Yocto-1.6

Objective: Learn how build a well tested cross-compiling toolchain using the eglibc C library

This is an optional exercise for home

If you only want to download a the result of this lab, go to the next chapter.

On a real fast machine like a Dell T7500 with 2 Xeon X5670 (2 x 6 cores/24 threads @ 2.93 GHz/96GB RAM), `bitbake` will run for an hour to complete the build.

On a Core-i7 Quad-Core laptop, like the Dell E6520 with Core i7 2760m/16GB RAM, you should expect 3-4 hours.

Expect a long, long time on a Core 2 Duo with small amount of RAM.

Yocto shows a stack of current executing tasks, as well as to total number of tasks, so you quickly get an idea about the build time.

After this lab, you will be able to:

- Generate a modern toolchain for the Beaglebone.

Setup

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

Install needed packages

Install the packages needed for this lab, if you havent done this before:

```
make prepare
```

Getting Yocto

```
git clone git://git.yoctoproject.org/poky poky-daisy
```

Then checkout the daisy branch (Yocto-1.6).

```
cd poky-daisy
git checkout -b daisy origin/daisy
```

Configuring Yocto

Once you have Yocto installed, you should configure it for your board.

```
cd poky-daisy
. oe-init-build-env build-beaglebone
```

This will create the `build-beaglebone` directory

Check the `build-beaglebone/conf` configuration directory.

An important file is `local.conf`

Configuring Yocto in local.conf

You should edit the `local.conf` to optimize for your own machine.

Edit the MACHINE variable to set it to the ”beaglebone”.

This will ensure that the cross compiler will build an ARMv7 toolchain with `NEON` support.

```
# There are also the following hardware board target machines included for
# demonstration purposes:
#
MACHINE ?= "beaglebone"
```

The default is to build a toolchain without libraries for static linking.

We will use static linking, so we need to add support by:

```
# Add libraries for static linking
#
IMAGE_INSTALL_append = " eglibc-staticdev"
SDKIMAGE_FEATURES += "staticdev-pkgs dev-pkgs"
```

A good place is right after the definition of `EXTRA_IMAGE_FEATURES`.

If you use a common download directory, you might want to change the DL_DIR variable.

```
# The default is a downloads directory under TOPDIR which is the build directory.
#
#DL_DIR ?= "${TOPDIR}/downloads"
```

If you have access with a DVD/USB memory with the tarballs, then you may want to copy those to the `build-beaglebone/downloads` directory to speed up the build.

Change the package mechanism to ipk

```
# Package Management configuration
#
# This variable lists which packaging formats to enable. Multiple package backends
# can be enabled at once and the first item listed in the variable will be used
# to generate the root filesystems.
# Options are:
#  - 'package_deb' for debian style deb files
#  - 'package_ipk' for ipk files are used by opkg (a debian style embedded package manag
#  - 'package_rpm' for rpm style packages
# E.g.: PACKAGE_CLASSES ?= "package_rpm package_deb package_ipk"
# We default to rpm:
PACKAGE_CLASSES ?= "package_ipk"
```


Building the Cross-Compiler

Yocto has the ability to generate a Software Development Kit (SDK) for an image.

By generating an SDK, you get a cross-compiler with everything needed to build further applications outside the Yocto build system.

The SDK contains all the header files for the applications and libraries included in the image. [2]

```
time bitbake core-image-minimal
```

followed by

```
time bitbake core-image-minimal -c populate_sdk
```

`core-image-minimal` is just that, so you may want to build a more complete image/toolchain.

```
time bitbake core-image-sato
```

followed by

```
time bitbake core-image-sato -c populate_sdk
```

The end result will be a script file in `~/felabs/sysdev/poky/build/tmp/deploy/sdk`

It is called something similar to:

```
poky-eglibc-x86_64-core-image-minimal-armv7a-vfp-neon-toolchain-1.6.sh
```

(The version number may differ)

[2] Some documentation recommends to do **bitbake meta-toolchain** or **bitbake meta-toolchain-sdk** but for some reason, the static libraries does not seem to be included when you do it this way

Installing a cross-compiling toolchain built by Yocto

Objective: Install a well tested cross-compiling toolchain using the eglibc C library

Getting the Yocto SDK

If you have done the Extra Lab: `Building a cross-compiling toolchain using Yocto`, you should have the file:

`poky-eglibc-x86_64-core-image-minimal-armv7a-vfp-neon-toolchain-1.6.sh`

or

`poky-eglibc-x86_64-core-image-sato-armv7a-vfp-neon-toolchain-1.6.sh`

or similar (the version number may differ)

If not, You can download the toolchain from `ftp://ftp.emagii.com/pub/training/tools` or copy if from the DVD/USB stick, if you got it from your teacher.

Install the latest version, and preferably `core-image-sato` over `core-image-minimal`.

Make sure this script is executable (Normally it should be). chnmod

Installing the Cross-Compiler

Caution: While the installation script allows you to install the toochain anywhere it will not work, unless you install it in "/opt/poky/1.6"

You normally do not have write access to `/opt` so you should create the install directory before you install the cross-compiler.

```
sudo mkdir -p /opt/poky
sudo chown ${USER} /opt/poky
```

Run the installation script.

You will need to fix LDFLAGS, otherwise it will cause problems with building U-Boot later.

Edit the `environment-setup-cortexa8hf-vfp-neon-poky-linux-gnueabi` file in the installation directory:

Comment away the existing definition of LDFLAGS and replace it with an empty defintion.

```
export LDFLAGS=""
```

From the install directory, copy the `environment-setup-cortexa8hf-vfp-neon-poky-linux-gnueabi` to `toolchain.sh` in the `~/felabs/sysdev` directory, and make sure it is executable.

If you are running any of the extra labs for creating a toolchain they may overwrite this file, so beware that you are not running the wrong cross-compiler. You could name it something else like `yocto-toolchain.sh`

Now you can use the cross compiler by just sourcing this file.

```
source toolchain.sh
```

Simplifying access to the toolchain script

For easier to access the toolchain script, you may want to create the `~/bin` directory, and copy the script here.

Add

```
PATH=~/bin:$PATH
```

to the end of `~/.bashrc`

Using the cross-compiler and sudo

Sometimes, the documentation may ask you do to:

```
sudo ${CROSS_COMPILE}<cmd>
```

This actually does not work, since when you enter superuser mode, you lose your current environment and CROSS_COMPILE becomes undefined.

If you run into problems with this, you need to

```
sudo su
source toolchain.sh
${CROSS_COMPILE}<cmd>
exit
```


Extra Lab: Installing the Codesourcery Lite toolchain

Objective: Setup the Mentor Sourcery Codebench Lite cross-compiling toolchain

This is an optional exercise for home

Some people like to use this toolchain, so it is included for reference. It will not be used during the labs.

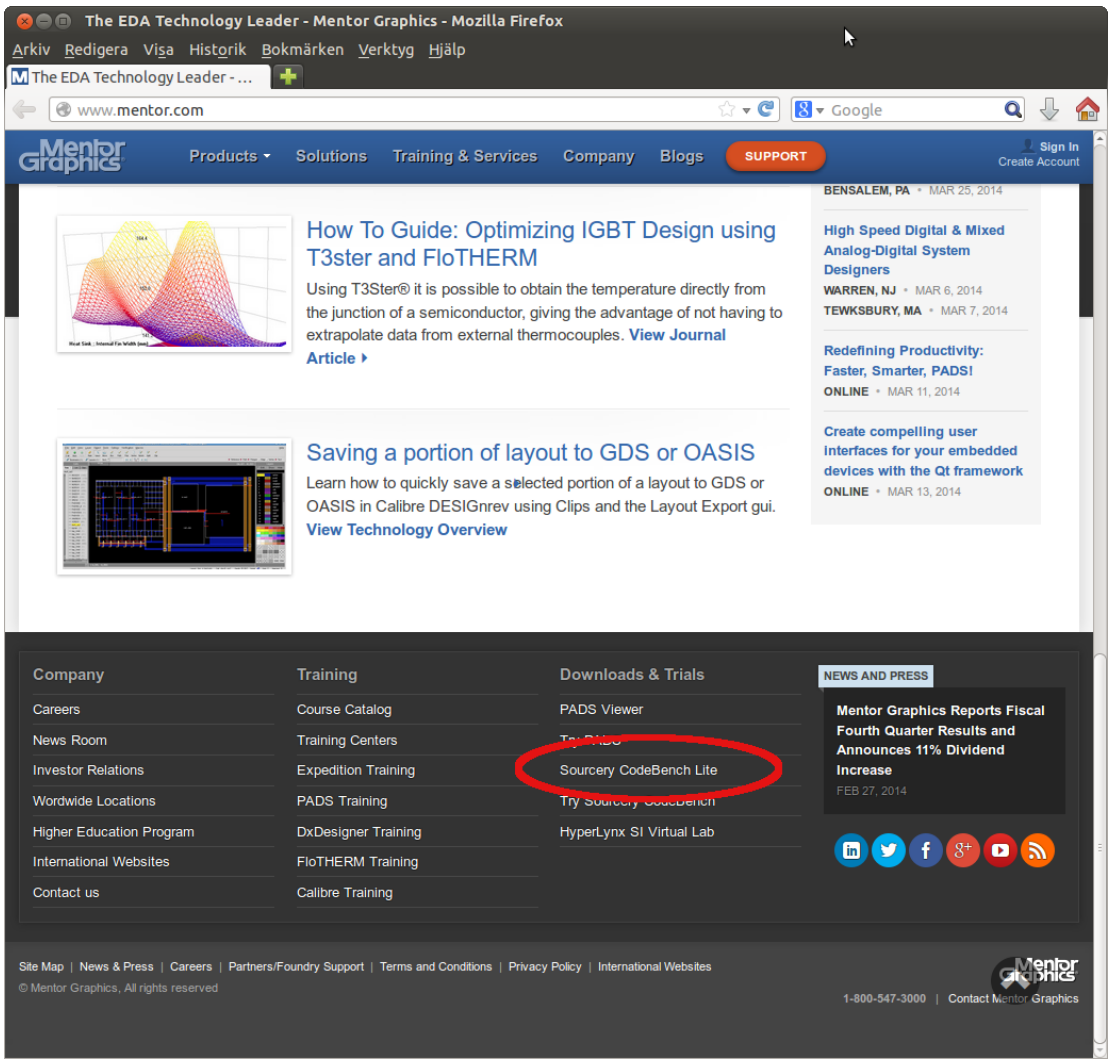
Sourcery Codebench Lite

The Sourcery Codebench Lite, is a free-of-charge high-quality toolchain, which is available with multiple library options.

There is a low cost commercial version, which contains an Eclipse Environment.

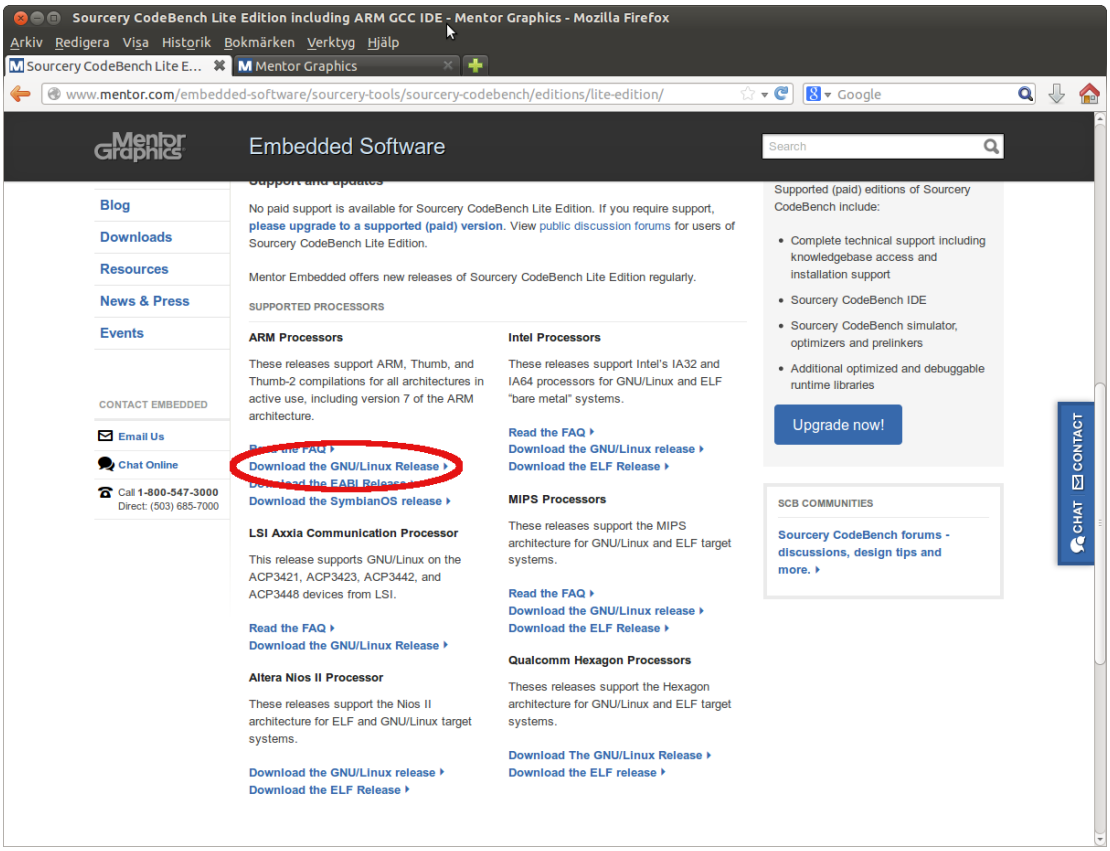
Getting the Toolchain

Open a Browser and go to the Mentor Graphics Homepage

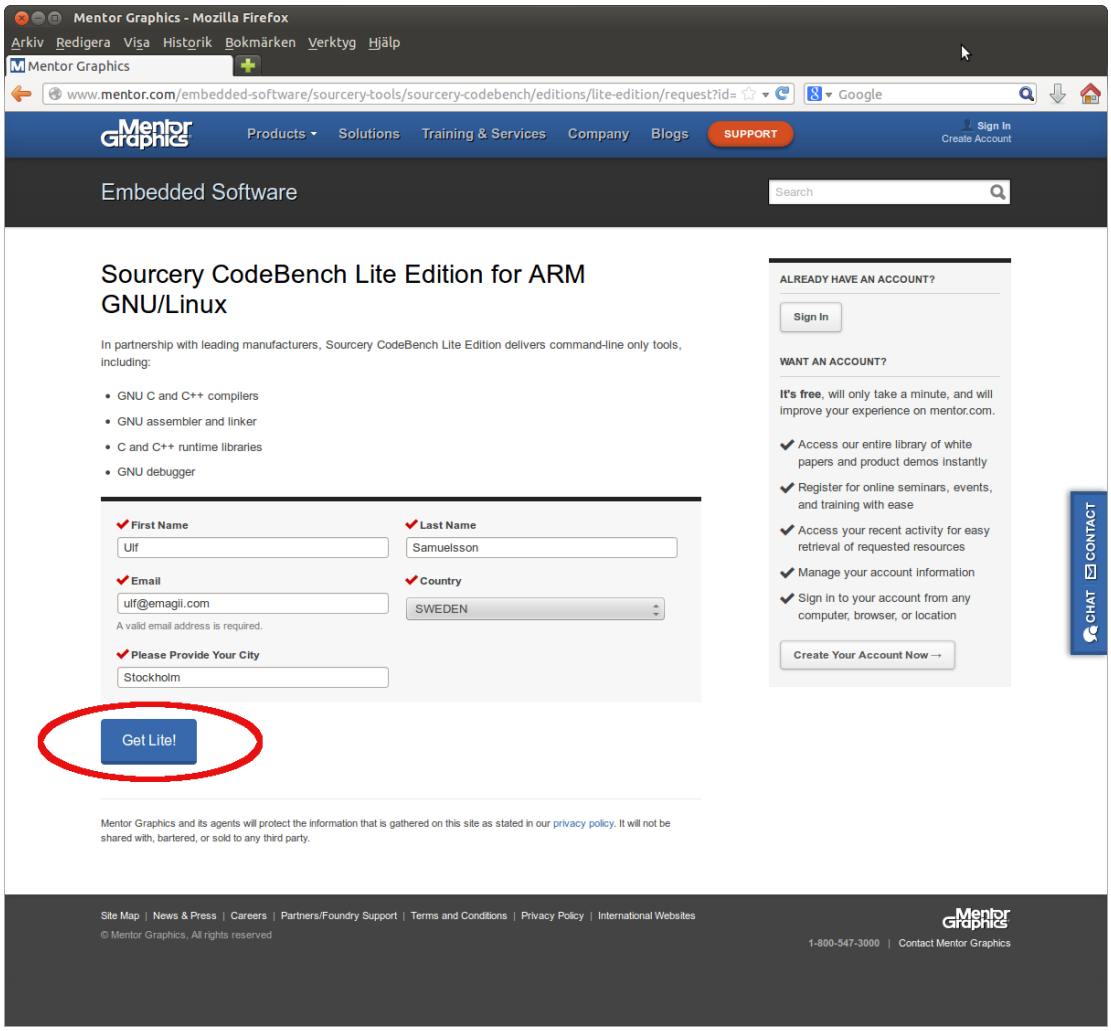


Go to the bottom and open the Sourcery Codebench Lite page

Once on the page, select the Download the Gnu/Linux Release for the ARM processor.

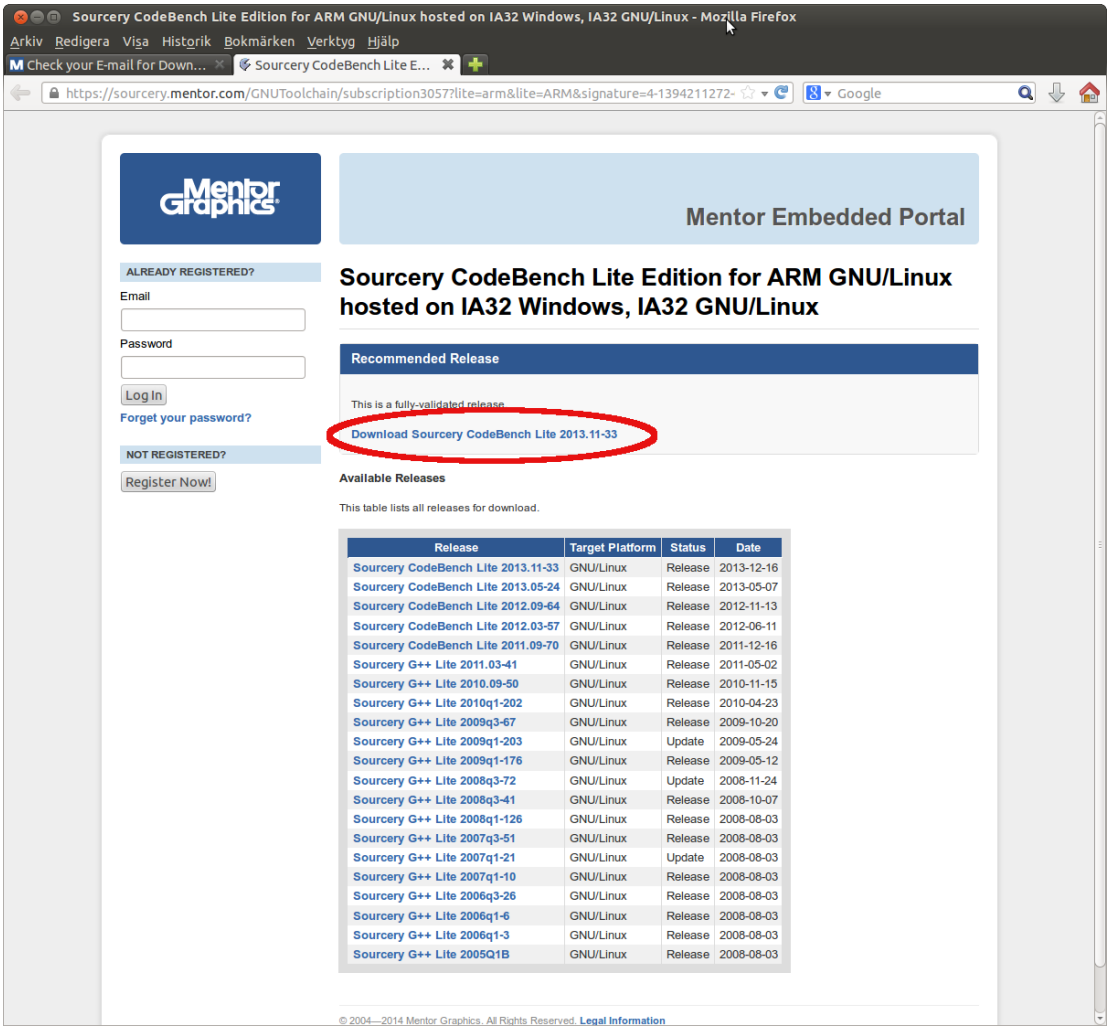


Fill in your personal details and click the **Get Lite!** button.

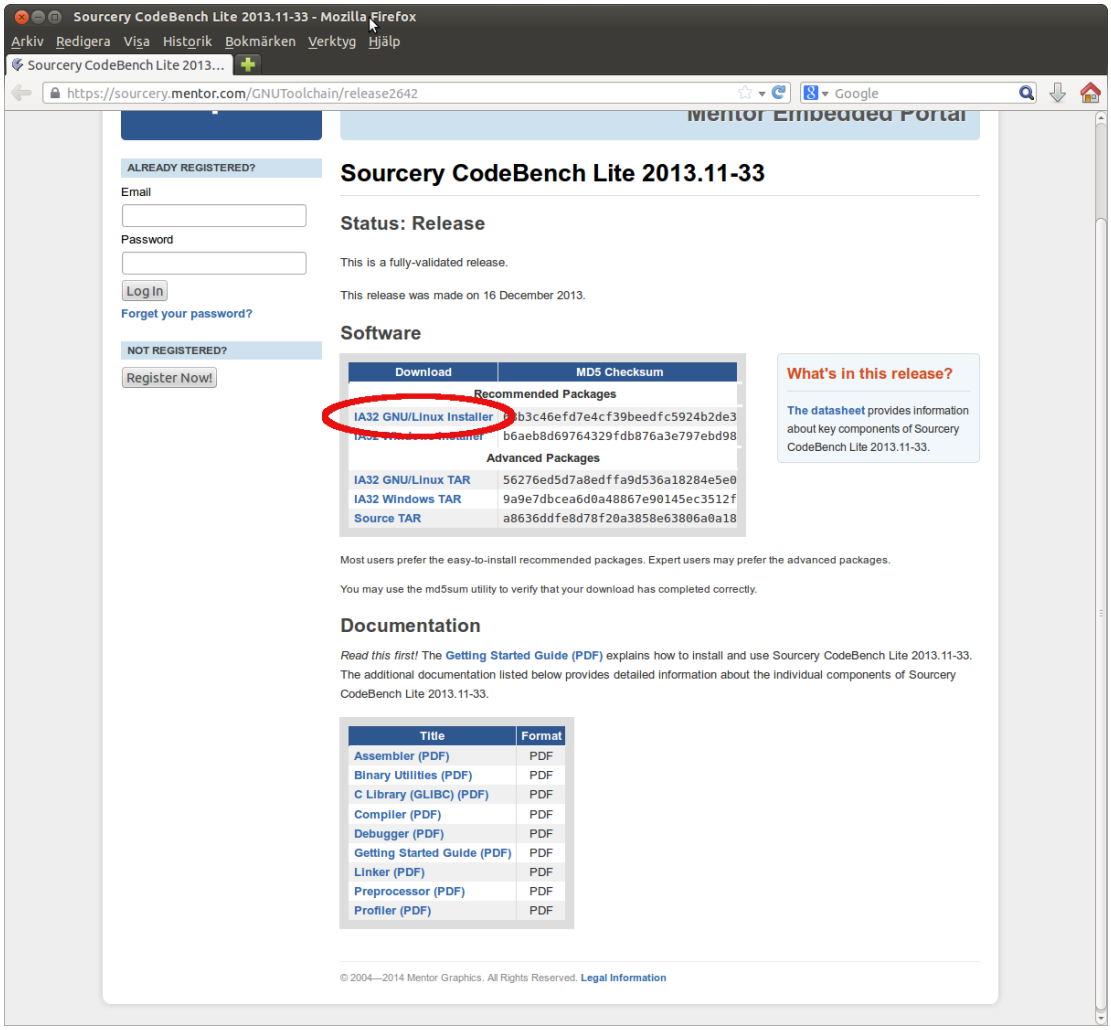


You will get a mail with the download location. Click on the link in the mail and you will open a page from where you can select your download.

Select the Sourcery Codebench Lite 2013.11-33 version



Right Click the **IA32 GNU/Linux Installer** and save at an appropriate place.



Getting-Started.pdf contains the installation guide and should also be downloads.

You may also want to download the rest of the documentation.

Installing the Sourcery Codebench Lite

Go to the directory where you downloaded the Installer and make it executable.

```
chmod a+x arm-2013.11-33-arm-none-linux-gnueabi.bin
```

Run the installer

```
./arm-2013.11-33-arm-none-linux-gnueabi.bin
```

The installer should be pretty obvious. If not, use the **Getting-Started.pdf** guide.

After the Installation, the path must be set up.

Edit "`~/.bashrc`" and add the path to the Installation.

```
export PATH=<install-dir>/bin:$PATH
```

You may also want to create a file `sourcery.sh` to source for setting up the toolchain.

```
#!/bin/sh
export ARCH=arm
export GCCROOT=<install-dir>
export PATH=$GCCROOT/bin:$PATH
export CROSS_COMPILE=arm-none-linux-gnueabi-
```

Make sure it is executable:

```
chmod a+x sourcery.sh
```

And then source it.

```
. ./sourcery.sh
```

Check the setup by checking the version of the C-Compiler.

```
arm-none-linux-gnueabi-gcc --version
```

Your output should be similar to:

```
arm-none-linux-gnueabi-gcc (Sourcery CodeBench Lite 2013.11-33) 4.8.1
Copyright (C) 2013 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Verify that the first line of the output contains: Sourcery CodeBench Lite 2013.11-33.

Whenever you need to run the Sourcery CodeBench Lite toolchain, you should source the `sourcery.sh` file to set up the environment.

Extra Lab: Simplified cross-compiler generation using Crosstool-NG

Objective: Get a toolchain based on the uClibc C library

Optional Lab for Home

The `crosstool-ng` project aims to produce a working toolchain for many architectures. It is starting to show its age, and does not fully support the Beaglebone, but is included as a reference.

The results will not be used during this training.

After this lab, you will have a:

- uclibc toolchain generated by the *crosstool-ng* tool

[3]

[3] The current generation of Crosstool-NG has problems with the ARMv7 architecture, which makes it hard to use with the Beaglebone for Linux

Setup

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

Clone the prepared scripts to build a toolchain

```
make toolchain
```

This will use git to clone a build environment which will use Crosstool-NG to build a cross-compiler for an ARMv7 chip.

The command executed will be:

```
git clone https://github.com/emagii/crosstool-ng-armv7a.git
```

The setup for an ARMv7 chip is not that simple yet, so the build has been automated.

The next **optional** lab will build the toolchain manually. This is an exercise left for only those interested. In real life, the toolchain will either be downloaded or built by a buildsystem like **Buildroot** or **Yocto** .

If you did not do this previously, install the packages needed for this lab:

(You will have to supply the super-user password)

```
make prepare
```

Build the toolchain

```
cd crosstool-ng-armv7a
make
```

If you are running this lab at home, then you just wait until the build completes. This can take anywhere from 15 minutes to an hour, depending on your machine.

If you are running the lab during a training, inform the trainer that your build has started.

Known issues

Source archives not found on the Internet

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

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

ppl-0.10.2 compiling error with gcc 4.7.1

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

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

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

Testing the toolchain

You can now test your toolchain. You need to setup the environment, and this is easy, just `source` the "`toolchain.sh`" script that was created during the build.

This looks like:

```
#!/bin/sh
export ARCH=arm
export GCCROOT=/usr/local/uclibc/arm-unknown-linux-uclibcgnueabihf
export PATH=$GCCROOT/bin:$PATH
export CROSS_COMPILE=arm-linux-
```

Go ahead and source it:

```
source toolchain.sh
```

You should also copy the script to the toplevel directory for use in later labs.

```
cp toolchain.sh ..
```

First test if you can access the compiler by checking its version:

```
arm-linux-gcc --version
```

It should write out something similar to:

```
arm-linux-gcc (crosstool-NG 1.19.0) 4.8.1
Copyright (C) 2013 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

There is a Hello World test program in the `example` directory and you can try compiling this with the arm-linux-gcc compiler.

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

Cleaning up

Note: Do not do the cleanup right now, since some files will be needed for a later lab.

To save about 4.3 GB of storage space, do a `./ct-ng clean` in the Crosstool-NG source directory. This will remove the source code of the different toolchain components, as well as all the generated files that are now useless since the toolchain has been installed in `/usr/local/uclibc`.

The source files are located in `$HOME/cross/src`

Extra Lab: Building a cross-compiling toolchain using Crosstool-NG

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

Optional Lab for Home

The `crosstool-ng` project aims to produce a working toolchain for many architectures. It is starting to show its age, and does not fully support the Beaglebone, but is included as a reference.

The results will not be used during this training.

Introduction

After this lab, you will be able to:

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

[4]

Setup

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

Install needed packages

Install the packages needed for this lab:

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

Getting Crosstool-ng

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

[4] The current generation of Crosstool-NG has problems with the ARMv7 architecture, which makes it hard to use with the Beaglebone for Linux

Installing Crosstool-ng

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

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

Then you can get Crosstool-ng help by running

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

Configure the toolchain to produce

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

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

Unfortunately, there is not a good configuration for the Beaglebone available in Crosstool-NG.

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

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

This generates the `.config` file in the source directory, but we will replace it with something useful.

Copy the `uclibc-cortex-a8.config` and the `uClibc-0.9.33.config` files from the toolchain made easy lab, to the crosstool-NG top directory, and then copy `uclibc-cortex-a8.config` to the `.config` file.

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

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

In `Path and misc options`:

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

In `Toolchain options`:

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

In `Binary utilities`:

- Set `binutils version` to `2.23.1`.

In `Debug facilities`:

- Enable `gdb`, `strace`.

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

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

Produce the toolchain

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

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

Nothing is simpler:

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

And wait!

Known issues

Source archives not found on the Internet

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

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

ppl-0.10.2 compiling error with gcc 4.7.1

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

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

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

Testing the toolchain

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

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

Cleaning up

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

Bootloader - U-Boot

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

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

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

The **Beaglebone Black** board, which uses an AM335x Sitara processor, falls into the second category.

The Sitara processors support a multitude of boot sources, and the boot source is selected by configuring a set of pins at reset. The Beaglebone limits the choices to one of two options and you choose which one, with a button.

The monitor integrated in the ROM reads the internal eMMC flash chip to search for a valid bootloader. If the user button is pressed at reset, the Sitara processor will instead search for a micro-SD card inserted in the connector.

The U-Boot bootloader programmed into the Beaglebone Black boots linux by executing the contents of the "bootcmd" environment variable, which normally tries to use the SD-Card contents first, and the eMMC memory only if booting from SD-Card fails, so normally the button does not have to be pressed.

Setup

Go to the `~/felabs/sysdev` directory and the enter the `bootloader` subdirectory.

U-Boot setup

Download U-Boot from the mainline U-Boot download site:

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

We want to figure out what is going on, so we will init a git repo.

Then we will add all the files and create a commit.

```
git init
git add .
git commit -m "Initial Commit" -s
```

The '-m' switch will tell git to use the following string as the commit message, and the '-s' switch will sign the commit with your user info as defined above.

We will apply two patches that are in the `data` directory that needs to be applied.

- `0001-arm-omap-i2c-don-t-zero-cnt-in-i2c_write.patch` This is a minor fix for a device driver.
- `0002-Read-environment-from-uSetup.txt-at-boot-if-present.patch` This is a fix which adds a significant functionality to the bootloader and is neccessary for the labs.

The traditional way is to '`cat`' the patch and pipe it through the '`patch`' utility, but we will do it using git instead

Example: Applying a patch in the traditional Way:

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

Example: applying a patch using git:

```
git am /path/to/0001-arm-omap-i2c-don-t-zero-cnt-in-i2c_write.patch
```

'`git am`' will not only apply the patch, it will also commit the patch using the original patch message. There are more requirements on a patch when you use "git am", since it will only accept well formed patches.

If a patch fails, then it will give you different options on how to handle the problem.

Apply the patch.

```
cd u-boot-2013.10
git am ../data/*.patch
```

Patching with "git am' should succeed, if you did not patch using the traditional way first.

If it fails, you can abort the patch through:

```
git am --abort
```

You can check the status of the git tree. Try:

```
git status
```


If you see any modified files, you can restore them using the checkout command.

```
git checkout <filename>
```

This will restore the file to its original state.

Try out:

```
git log
```

U-Boot Source Tree

Get an understanding of U-Boot's configuration and compilation steps by reading the `README` file, and specifically the *Building the software* section.

Basically, you need to:

- set up the cross compiler (You could source the `toolchain.sh` script in the top directory) or if you use the Yocto toolchain, source the appropriate `environment...` file

  ```
  #!/bin/sh
  export ARCH=arm
  export GCCROOT=/usr/local/uclibc/arm-unknown-linux-uclibcgnueabihf
  export PATH=$GCCROOT/bin:$PATH
  export CROSS_COMPILE=arm-linux-
  ```

- Configure U-Boot to build for the Beaglebone Black

  Note that for our platform, the configuration file is

  `include/configs/am335x_evm.h`

  Read this file to get an idea of how a U-Boot configuration file is written;

  Then configure U-Boot.

  ```
  $ make am335x_boneblack_config
  Configuring for am335x_boneblack - Board: am335x_evm, Options: SERIAL1,CONS_INDEX=1,
  ```

  If you get a lot of errors, try `make clean` or `make distclean` and rerun.

  You should see something similar to:

  ```
  Configuring for am335x_boneblack - Board: am335x_evm,
      Options: SERIAL1,CONS_INDEX=1,EMMC_BOOT
  ```

  This configuration step generates a `Makefile` fragment which you may want to look at.

  `include/autoconf.mk`

- Finally, run `make`[5], which should build U-Boot.

One would think that the build generated a number of new files. Checking with:

```
git status
```

should print out:

```
# On branch master
nothing to commit (working directory clean)
```

If you do an `ls` you **will** see some new files like `u-boot.img`.

The reason they are not detected, is that they are hidden by the `.gitignore` file filter. If you name a `regexp` in `.gitignore` it will be ignored.

List the `.gitignore` by `more .gitignore`.

[5] 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.

Rescue binaries

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

The U-Boot boot process

When U-Boot starts, it loads the environment from a non-volatile memory to RAM The environment is a set of variables, which can be used as scripts through the `run` command. Other environment variables are used to define constants like addresses, for use in script variables

The location of the environment memory is defined in a compile time constant, typically set in `board.cfg`. Examples of environment memory types are NOR Flash, NAND Flash, SPI Flash and I2C EEPROM.

The first time U-Boot is started on a board, it will load a default environment.

You can modify a variable or create a new one with the `setenv` command.

If you have an environment memory, you can save the updated environment with the `saveenv` command.

The next time, U-boot is started, it will detect that you have a valid environment, and load this, instead of the default environment.

Not every board has an environment memory, and an example of the this is the **Beaglebone Black** . Such board will always load the default memory at start-up.

The U-Boot process depends on two special variables `bootcmd` and `bootargs`.

If the user does not intervene, by pressing `<return>`, U-Boot will `run` the `bootcmd` variable script. Typically the last command executed in `bootcmd` is a command which boots the linux kernel. There are several commands available.

- `bootm` Boots a `uImage`
- `bootz` Boots a `zImage`

When any of the boot commands are executed, they will boot the kernel and pass the contents of the `bootargs` variable to the kernel.

The way to use the U-Boot environment has been refined over time. In early days, you would typically set the `bootargs` directly. The current approach is to run a variable script which generates `bootargs` from a set of other variables.

More information about U-Boot is available in `http://www.denx.de/wiki/DULG/Manual`

The **Beaglebone Black** predefined environment is set-up to boot from an SD-Card.

The **Beaglebone Black** `bootcmd` is fairly complex, and tries to detect an external SD-Card. If not present, it will use the soldered SD-Card. Once the card is selected, U-Boot will try to load the file `uEnv.txt` and import the contents into the environment.

The syntax for setting variables in `uEnv.txt` are

```
<VARIABLE>=<VALUE>
```

An Example:

```
uenvcmd=run tftp_kernel tftp_dtb netargs bootkernel
```

This creates the variable script `uenvcmd` to run four scripts in succession.

The variable `uenvcmd` has a special meaning in the **Beaglebone Black** U-Boot. Once the `uEnv.txt` has been imported, `bootcmd` will test for the precense of this variable, and if it exists, it will be `run`.

By setting this to a suitable value, you can boot from any source.

If `uenvcmd` is not defined in `uEnv.txt`, U-Boot will try to load the kernel and device-tree file from the second partition of the SD-Card which normally should contain the file system. The default assumes that they are located in the `/boot` directory.

It will set-up the `bootargs` variable to use the second partition of the SD-Card as its root and then boot the kernel.

There is a small problem with this approach, and that it does not give the user any flexibility. If the boot is stopped before `bootcmd` is executed, only the default environment is available.

If `bootcmd` is executed, it cannot easily be stopped, so while U-Boot can be booted from any source, it boot sequence cannot be changed without modifying the `uEnv.txt` file.

The patched U-Boot used in this lab, supports an alternative method.

The U-Boot modified boot process

After U-Boot has loaded the default environment, it will load the `uSetup.txt` file from the seclected SD-Card and import its content into the environment. It has the same syntax as the `uEnv.txt` file. This happens before U-Boot checks if the user wants to abort the automatic boot, by pressing `<return>` key.

If the user does not abort, the `bootcmd` command will be executed.

By changing the `bootcmd` command in `uSetup.txt`, you can make U-Boot boot from any source.

If the user aborts the automatic boot, and enters the U-Boot command loop, `uSetup.txt` can have created scripts, that can be run manually, to allow easy selection of the boot sequence.

Preparing an SD-card with U-Boot

Copy the generated `MLO` and `u-boot.img` files to the SD card FAT partition. `MLO` is the first stage bootloader, `u-boot.img` is the second stage bootloader.

Preparing an SD-card with U-Boot using uEnv.txt

We will create a new `uEnv.txt` on the host using the template below.

Change values to fit your setup:

You should use the same values for **ipaddr** and **serverip** as used in the toplevel `host.mk` file.

Caution: For technical reasons, long lines are split up in this pdf file, using the '\' character. In the `uEnv.txt` file, they must be a single line with the '\' character removed.

```
tftp_dtb=tftp ${fdtaddr} \
  am335x-boneblack.dtb
```

should look like

```
tftp_dtb=tftp ${fdtaddr} am335x-boneblack.dtb
```

in the file.

Variables using split lines like this are 'netargs'

```
ipaddr=192.168.0.100
serverip=192.168.0.1
loadaddr=0x80200000
fdtaddr=0x80F80000
IMAGE=rootfs
console=ttyO0,115200n8
nfsopts=nolock
netargs=setenv bootargs console=${console} root=/dev/nfs \
  nfsroot=${serverip}:/tftpboot/${IMAGE},${nfsopts} rw ip=${ipaddr}
tftp_kernel=tftp ${loadaddr} zImage
tftp_dtb=tftp ${fdtaddr} am335x-boneblack.dtb
bootkernel=bootz ${loadaddr} - ${fdtaddr}
uenvcmd=run tftp_kernel tftp_dtb netargs bootkernel
mmcboot=echo MMC boot disabled
nandboot=echo NAND boot disabled
```

Caution: In `ttyO2`, it's the capital letter `O`, like in OMAP and not the number zero)

Copy the file to the FAT partition.

Create an empty file `uSetup.txt` in the same place.

Remember to run `sudo sync` before unmounting and removing the card.

Preparing an SD-card with U-Boot using uSetup.txt

Our U-Boot patch (which is not standard) will start by reading the `uSetup.txt` before `bootcmd` is executed, so you have a chance to stop the boot, by pressing **return**.

Below is a template for the file `uSetup.txt`, but it needs to be edited before it can be used.

Create a `uSetup.txt` file on your host.

You should use the same values for **ipaddr** and **serverip** as used in the toplevel `host.mk` file.

Caution: For technical reasons, long lines are split up in this pdf file, using the '\' character. In the `uEnv.txt` file, they must be a single line with the '\' character removed.

The assignments of 'netargs' and 'mmcargs' should each be a single line.

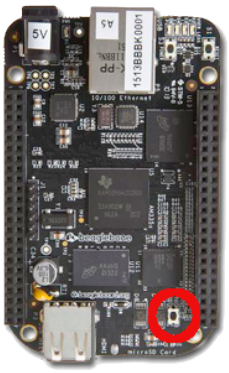
In `ttyO2`, it's the capital letter `O`, like in **O**MAP and not the number zero)

```
ipaddr=192.168.0.100
serverip=192.168.0.1
loadaddr=0x80200000
fdtaddr=0x80F80000
IMAGE=rootfs
console=ttyO0,115200n8
nfsopts=nolock
netargs=setenv bootargs console=${console} root=/dev/nfs \
  nfsroot=${serverip}:/tftpboot/${IMAGE},${nfsopts} rw ip=${ipaddr}
tftp_kernel=tftp ${loadaddr} zImage
tftp_dtb=tftp ${fdtaddr} am335x-boneblack.dtb
bootkernel=bootz ${loadaddr} - ${fdtaddr}
mmcargs=setenv bootargs console=${console} ${optargs} root=${mmcroot} \
  rootfstype=${mmcrootfstype} rootwait
mmcrootfstype=squashfs
nandboot=echo NAND boot disabled
bootcmd=run tftp_kernel tftp_dtb netargs bootkernel
```

Do not copy `uSetup.txt` to the SD-Card yet, instead keep the empty `uSetup.txt` in the FAT partition.

Testing U-Boot on the MMC card

Insert the MMC card into the Beaglebone Black board, Unplug the power cord. While pressing the user button, reinsert the power cord and check that it boots your new bootloader. You can verify this by checking the build date: (Should obviously be todays date). Press the **RETURN** button, when the prompt says `Hit anykey to stop autoboot:`



If the SD-Card boot does not work, the processor will try to boot from the internal eMMC memory, and the date will be different from todays date.

If that happens, retry removing the power plug, then push the button, reinsert power.

If it still does not work , verify that the SD card is correctly setup using `cfdisk`

- heads=255
- sectors=63
- partition 1 is bootable
- partition 1 is FAT formated and contains `MLO`, `u-boot.img` and `uEnv.txt`

If it works you will see something similar to:

```
U-Boot SPL 2013.10-gbd39439-dirty (Mar 03 2014 - 18:03:52)
spl: error reading image args, err - 0
reading u-boot.img


U-Boot 2013.10-gbd39439-dirty (Mar 03 2014 - 20:04:45)

I2C:   ready
DRAM:  512 MiB
WARNING: Caches not enabled
MMC:   OMAP SD/MMC: 0, OMAP SD/MMC: 1
Using default environment

Net:   <ethaddr> not set. Validating first E-fuse MAC
cpsw, usb_ether
Hit any key to stop autoboot:  0
```

The message `reading u-boot.img` also confirms that U-Boot has been loaded from the MMC device. The error message is due to the `Falcon mode` which will allow booting the kernel directly from `MLO` without loading U-Boot. We do not use this mode.

When U-Boot initializes, it will execute the contents of the `bootcmd` variable as a script.

Before that happens, it will wait for a few seconds (value of `bootdelay`) giving the user a chance to stop the boot.

Press the `<return>` button before the counter expires, to enter the U-Boot shell.

(You may have to reset the Beaglebone if you did not manage to press the `<return>` button in time)

```
U-Boot #
```

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

You can get more help on a specific command. Try `help boot`.

The Default U-Boot Environment

U-Boot supports an environment, and you can extend the commandset by defining environment variables. These can be "used" as scripts through the `run` command It is possible to define a compile time environment, which can be extended at run time. On the Beaglebone Black, the environment can be updated from the file uEnv.txt stored together with `MLO` and `u-boot.img` on the SD-Card.

All environment variables are normally displayed in alphabetical order, when printed. To make it easier to understand, they are displayed sorted a little more logical below.

```
arch=arm
baudrate=115200
board=am335x
board_name=A335BNLT
board_rev=0A5C
boot_fdt=try
bootdelay=1
bootdir=/boot
bootenv=uEnv.txt
bootfile=zImage
bootpart=1:2
console=ttyO0,115200n8
cpu=armv7
ethact=cpsw
ethaddr=c8:a0:30:c4:4e:8f
fdt_high=0xffffffff
fdtaddr=0x80F80000
fdtfile=am335x-boneblack.dtb
filesize=4e
loadaddr=0x80200000
mmcdev=1
mmcroot=/dev/mmcblk0p2 ro
mmcrootfstype=ext4 rootwait
nfsopts=nolock
optargs=quiet drm.debug=7
ramroot=/dev/ram0 rw ramdisk_size=65536 initrd=${rdaddr},64M
ramrootfstype=ext2
rdaddr=0x81000000
rootpath=/export/rootfs
soc=am33xx
spibusno=0
spiimgsize=0x362000
spiroot=/dev/mtdblock4 rw
spirootfstype=jffs2
spisrcaddr=0xe0000
static_ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}::off
stderr=serial
stdin=serial
stdout=serial
usbnet_devaddr=c8:a0:30:c4:4e:8f
vendor=ti
ver=U-Boot 2013.10-gbd39439-dirty (Mar 03 2014 - 20:04:45)
```


The most important environment variable is the `bootcmd` script which is used to boot linux.

The Beaglebone Black will first determine which Device Tree file to use, based on the `$board_name` variable, then it will try to run mmcboot first on the SD-Card, then on the internal eMMC. finally if both fails, it will try nandboot (which does not work, since the board does not have a NAND flash)

```
bootcmd=
        run findfdt;
        run mmcboot;
        setenv mmcdev 1;
        setenv bootpart 1:2;
        run mmcboot;
        run nandboot;

findfdt=
        if test $board_name = A335BONE; then
                setenv fdtfile am335x-bone.dtb;
        fi;
        if test $board_name = A335BNLT; then
                setenv fdtfile am335x-boneblack.dtb;
        fi;
        if test $board_name = A33515BB; then
                setenv fdtfile am335x-evm.dtb;
        fi;
        if test $board_name = A335X_SK; then
                setenv fdtfile am335x-evmsk.dtb;
        fi;
        if test $fdtfile = undefined; then
                echo WARNING: Could not determine device tree to use;
        fi;
```

The normal way of booting is using the internal eMMC or the SD-Card, and this function is performed by `mmcboot`. The `mmcdev` variable is used to determine the bootsource.

The `bootpart` is used to determine the location of the kernel and device tree file. For the Beaglebone, they normally are located inside the root file system, in the `/boot` directory.

The root file system is normally in partition 2 of the SD–card, but we have not created a root file system yet, and certainly not loaded a root file system to the SD-Card.

```
mmcboot=
        mmc dev ${mmcdev};
        if mmc rescan; then
                echo SD/MMC found on device ${mmcdev};
                if run loadbootenv; then
                        echo Loaded environment from ${bootenv};
                        run importbootenv;
                fi;
                if test -n $uenvcmd; then
                        echo Running uenvcmd ...;
                        run uenvcmd;
                fi;
                if run loadimage; then
                        run mmcloados;
                fi;
        fi;

loadramdisk=load mmc ${mmcdev} ${rdaddr} ramdisk.gz

loadbootenv=load mmc ${mmcdev} ${loadaddr} ${bootenv}

importbootenv=
        echo Importing environment from mmc ...;
        env import -t $loadaddr $filesize

loadimage=load mmc ${bootpart} ${loadaddr} ${bootdir}/${bootfile}
```

The `mmcboot` code will first select the correct MMC device according to `mmcdev`. The MMC device will be initialized through the `mmc rescan` command. If the device is found, then the file `uEnv.txt` will be loaded and the contents will overlay the current environment, so any environment variable can be changed by redefining it in the uEnv.txt file. The changes are not persistent between reboots.

If you want to change the boot mechanism, then set the `uenvcmd` variable, because this will be executed, if it exists. A typical use was to change the kernel from uImage to zImage.

if `uenvcmd` does not boot the kernel, U-Boot tries to load the kernel from the root file system in the `/boot` directory, so the rootfs must contain the kernel.

Finally, the the kernel will be booted, using a device tree file. This is explained on the next page.

If the kernel requires a device tree file (which it does for the Beaglebone Black) then this is loaded from the `/boot` directory.

The Beaglebone Black is using the `am335x-boneblack.dtb` file.

`bootz` is used to boot a zImage

```
mmcargs=setenv bootargs console=${console} ${optargs} root=${mmcroot} \

         rootfstype=${mmcrootfstype}

loadfdt=load mmc ${bootpart} ${fdtaddr} ${bootdir}/${fdtfile}

mmcloados=
        run mmcargs;
        if test ${boot_fdt} = yes || test ${boot_fdt} = try; then
                if run loadfdt; then
                        bootz ${loadaddr} - ${fdtaddr};
                else
                        if test ${boot_fdt} = try; then
                                bootz;
                        else
                                echo WARN: Cannot load the DT;
                        fi;
                fi;
        else
                bootz;
        fi;
```

NFS Booting can be of interest, and this is supported by the Beaglebone default environment An easy way to boot from the network, is to set the `uenvcmd` variable to `run netboot`

```
netargs=setenv bootargs console=${console} ${optargs} root=/dev/nfs \

        nfsroot=${serverip}:${rootpath},${nfsopts} rw ip=dhcp

netboot=
        echo Booting from network ...;
        setenv autoload no;
        dhcp;
        tftp ${loadaddr} ${bootfile};
        tftp ${fdtaddr} ${fdtfile};
        run netargs;
        bootz ${loadaddr} - ${fdtaddr}
```

Finally, the environment supports DFU (Device Firmware Upgrade), SPI Boot and Booting a RAMdisk, but this is not used on the Beagleboard Black.

```
dfu_alt_info_emmc=rawemmc mmc 0 3751936
dfu_alt_info_mmc=
        boot part 0 1;
        rootfs part 0 2;
        MLO fat 0 1;
        MLO.raw mmc 100 100;
        u-boot.img.raw mmc 300 400;
        spl-os-args.raw mmc 80 80;
        spl-os-image.raw mmc 900 2000;
        spl-os-args fat 0 1;
        spl-os-image fat 0 1;
        u-boot.img fat 0 1;
        uEnv.txt fat 0 1
dfu_alt_info_ram=
        kernel ram 0x80200000 0xD80000;
        fdt ram 0x80F80000 0x80000;
        ramdisk ram 0x81000000 0x4000000

ramargs=setenv bootargs console=${console} ${optargs} root=${ramroot} rootfstype=${ramro
ramboot=
        echo Booting from ramdisk ...;
        run ramargs;
        bootz ${loadaddr} ${rdaddr} ${fdtaddr}

spiargs=setenv bootargs console=${console} ${optargs} root=${spiroot} rootfstype=${spiro
spiboot=
        echo Booting from spi ...;
        run spiargs;
        sf probe ${spibusno}:0;
        sf read ${loadaddr} ${spisrcaddr} ${spiimgsize};
        bootz ${loadaddr}
```


Setting up the Beaglebone Black Networking in U-Boot

The networking commands in U-Boot typically needs to have `ipaddr` and `serverip` configured. The `uEnv.txt` we use, sets up these values.

Try:

```
print serverip
```

You will get an error message.

```
## Error: "serverip" not defined
```

The reason is that `uEnv.txt` is only read when the `bootcmd` variable is run.

Since you stopped U-Boot by pressing **return**, bootcmd did not run, and therefore `uEnv.txt` was not read and added to the environment.

Try resetting the board, by giving the command

```
reset
```

Check that the U-Boot date is still correct.

This time, do not press **return** to stop the boot. This time, `bootcmd` **is** executed, and `uEnv.txt` settings are added to the environment. We setup the environment to intentionally fail, so you will still get a prompt.

Again:

```
print serverip
```

`serverip` should now be set to your configuration.

Check the other network parameters.

```
print ipaddr
print ethaddr
```

If the MAC address is not set, you also need to set it in U-boot, If this is the case, please contact the teacher and you will be allocated a number XX. [6]

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

If you changed `ethaddr`, switch your board off and on again and let it run bootcmd.[7]. Don't forget to press the **user** button.

If `ipaddr` is not set, or not reasonable, set it manually

```
setenv ipaddr 192.168.0.100
```

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
```

[6] The **Beaglebone Black** contains an EEPROM which is programmed with the MAC address at production time, so you should not have this problem

[7] 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.

Saving the updated U-Boot environment

If you had a NAND flash, then you could have saved the environment by

```
saveenv
```

Since we are running from an MMC card, you have to retype this every time you reboot, or add it to your `uEnv.txt`.[8]

Note the difference in syntax.

U–Boot :

```
setenv <VAR> <VALUE>
```

uEnv.txt :

```
<VAR>=<VALUE>
```

Testing the customized U-Boot environment

You can then test the TFTP connection. First, put a small text file called `testfile.txt` in the directory exported through TFTP on your development workstation (Normally `/tftpboot`, and you may already have the file there).Then, from U-Boot, do:

```
tftp 0x80000000 testfile.txt
```

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

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

or

```
sudo /etc/init.d/xinetd restart
```

depending on which tftpd server you use.

You may in severe cases, need to reboot your host. If you still fail, please recheck your configuration.

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

```
md 0x80000000
```

[8] `ethaddr` can be set, but not changed, once saved unless the environment is erased by using an external programmer

Testing booting with environment in uSetup.txt

Replace the empty `uSetup.txt` file you previously created in the FAT partition with the one you created on the host. `sync`, unmount, and reinsert into the **Beaglebone Black** .

reset, (again check the date of the U-Boot) and stop the autoboot using `<return>`.

Again test the networking environment by checking the value of `ipaddr` and `serverip`

They should now be set.

Test one of our new commands.

```
print bootargs
run netargs
print bootargs
```

The netargs script have update the `bootargs` variable.

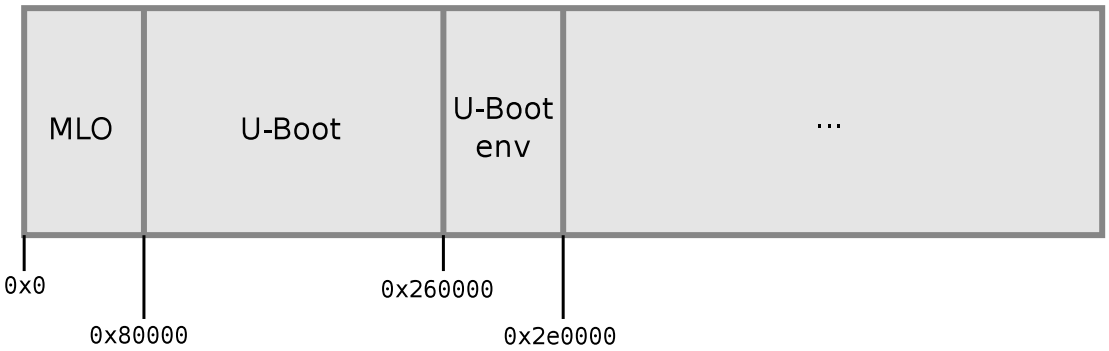
If `bootargs` has the value `console=${console} root=/dev/nfs` and nothing more, then you have not merged the line assigning the value to netargs with the next line.

Please edit `uSetup.txt` and fix the problems.

Reflashing from U-boot (Skip)

Since the Beaglebone Black does not have any NAND Flash, the following chapter is for reference only, and you can just read through it.

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



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

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

```
nand erase.chip
```

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

```
mmc rescan
```

This initializes the MMC interface.

```
fatload mmc 0 80000000 MLO
```

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

```
nandecc hw
```

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

```
nand erase 0 80000
```

This command erases a 0x80000 byte long space of NAND flash from offset 0[9].

```
nand write 80000000 0 80000
```

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

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

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

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

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

```
nand erase 260000 80000
```

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

You should now see the U-Boot prompt:

```
U-Boot #
```


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

Goto the `~/felabs/sysdev` and from there to the `kernel` subdirectory.

Get the sources

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

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

We will use `linux-3.14.4`, which this lab was tested with.

If you want to practice the patch command later, download the full 3.14 sources instead of the 3.14.4. Unpack the archive, which creates a `linux-3.14` directory. Remember that you can use `wget <URL>` on the command line to download files.

Enter the kernel source directory and create a git repo through

```
git init
git add .
git commit -m "Initial Commit" -s
```


Apply patches

Download the patch file corresponding to the latest 3.14.4 release: Put the patch file in the parent directory of the kernel source. This will be used to update the kernel from 3.14 to 3.14.

Patches are applied using the `patch` program which normally reads from `stdin`, but you can supply a filename using the `-i` switch.

Normally you have to supply a `-p<num>` switch. The patches contains filenames which are relative to a top directory. The `-p<num>` switch will shave off part of the directory structure of a patch filename. I.E:

If `-p1` is supplied, `/a/lib/mylib.c` becomes `/lib/mylib.c`

If `-p2` is supplied, `/a/lib/mylib.c` becomes `mylib.c`

Normally kernel patches requires `-p1` so a way to apply patches would be:

```
patch -p1 -i path/to/patch-x.y.z
```

Another way would be

```
cat path/to/patch-x.y.z | patch -p1
```

The downloaded patch is compressed. There is a way to apply a patch without uncompressing it first into a temporary file. Can you figure out how?[10]

Without uncompressing it first (!), apply the patch to the Linux source directory.

Use `git` to first add all files, and then commit the patch with a reasonable comment.

Check your repo with `git status` and `git log`. Does it look reasonable?

You can regenerate a patch from the repo. Try:

```
git format-patch -1
```

This will create a file `0001-<patch-name>.patch`

View the patch file with `vi` or `gvim` or any other editor, to understand the information carried by such a file. How are added or removed files described?

Remove the patch file after inspection.

If you selected to patch the 3.14 kernel, rename the `linux-3.14` directory to `linux-3.14.4`.

[10] Think zcat, bzcat or xzcat

Kernel - Cross-compiling

Objective: Learn how to cross-compile a kernel for a Beaglebone Black board.

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 **Beaglebone Black** board
- Use U-Boot to download the kernel
- Check that the kernel you compiled starts the system

Setup

Go to the `~/felabs/sysdev` and from there to the `kernel` subdirectory.

Install the required packages if you did not run `make prepare` before:

```
sudo apt-get install libqt4-dev u-boot-tools
```

`libqt4-dev` is needed for the `xconfig` kernel configuration interface, and `u-boot-tools` is needed to build an `uImage` kernel image file for U-Boot. We will use the more modern `zImage` so it might not be needed.

You should remove libqt3 development packages, using Synaptic if already present.

Target system

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

Kernel sources

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

Linux kernel configuration

Find the proper Makefile target to configure the kernel for the **Beaglebone Black** board (hint: the default configuration is not named after the board, but is related to the CPU name=AM33XX). The configuration files are in `arch/arm/configs`. Find a suitable configuration, and check with the teacher, if it is OK.

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

```
ARCH=arm make <board>_defconfig
```

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

This may fail, if libqt3 is installed and used during make. Deinstall libqt3 headers and retry.

Another reason is that `PKG_CONFIG_PATH` is not set up.

```
export PKG_CONFIG_PATH=/usr/lib/x86_64-linux-gnu/pkgconfig
```

Caution: The `toolchain.sh` file changes some critical things, which breaks the kernel configuration make targets. You may be able to 'make xconfig' by using a eparate terminal where you did not source 'toolchain.sh'.

If you have sourced the file, qt development libraries will not be found.

If you get mystic error messages regarding `qt-mt.pc` you may have to reset by `make distclean` and redo:

```
make <board>_defconfig
```

In the kernel configuration:

- In the `System Type` menu, make sure that `TI AM33XX` is selected. Otherwise, you do not have the correct `<board>_defconfig` We will boot our kernel with a device tree for our board so technically speaking, you can leave this option enabled, and still boot using a *Device Tree*,
- As an experiment, let's change the kernel compression from Gzip to XZ. This compression algorithm is far more efficient than Gzip, in terms of compression ratio, at the expense of a higher decompression time. You find XZ in the `General Setup` menu.
- Make sure that SquashFS, ext3 and ext4 filesystems are supported.
- Make sure that the AM335x Framebuffer is supported
- Exit the configuration tool, and save the changes. The result is in the `.config` file.

Cross-compiling environment setup

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

Linux requires some environment variables to be set:

- `ARCH`
- `CROSS_COMPILE`

You can also specify them on the command line at every invocation of `make`, i.e:

```
make ARCH=... CROSS_COMPILE=... <target>
```

Since we created a script file doing this we should just do:

```
source ../toolchain.sh
```


Cross compiling

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

```
make -j <n> LOADADDR=0x80008000 zImage
```

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

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

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

Next you need to build the device tree files.

```
make dtbs
```

You can see the Device Tree `.dtb` files which got compiled. Find which `.dtb` file corresponds to your board.

The kernel support dynamically loaded modules, which needs to be built separately.

```
make modules
```

Install the modules in the `/tftpboot/rootfs` directory. We will not use them right now, but we will need them later.

```
make INSTALL_MOD_PATH=/tftpboot/rootfs modules_install
```

Check the contents of the `/tftpboot/rootfs` directory. There should be a `lib` directory.

Copy the `zImage` and DTB files to the /tftpboot directory exposed by the TFTP server.

Setting up serial communication with the board

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

```
$ picocom -b 115200 /dev/ttyUSB0
```

Once connected, reset the board, not forgetting to press the **user button**.

Press the **return** key to stop U-Boot from executing the `bootcmd`.

You should now see the U-Boot prompt:

```
U-Boot #
```


Load and boot the kernel using U-Boot

We will use TFTP to load the kernel image to the **Beaglebone Black** board:

You need give a number of commands to boot the kernel, Note that you may already have done this for some in the `uSetup.txt` file.

Check variables with `print`.

- On the target, set the networking information (use the real ip addresses)
  ```
  setenv serverip 192.168.0.1
  setenv ipaddr 192.168.0.100
  ```

  Make sure that the bootargs U-Boot environment variable is set (it could remain from a previous training session, and this could disturb the next lab). We will set it to use a ramdisk, but since we do not have the ramdisk, the kernel will abort.

  ```
  setenv bootargs console=ttyO0,115200n8 root=/dev/ram0 rw ramdisk_size=65536 \
    initrd=0x81000000,64M rootfstype=ext2
  ```

  Make sure the above line is a single line.

- On the target, load `zImage` from TFTP into RAM at address 0x80200000:
  ```
  tftp 0x80200000 zImage
  ```
- Now, also load the DTB file into RAM at address 0x82000000:
  ```
  tftp 0x80F80000 am335x-boneblack.dtb
  ```
- Boot the kernel with its device tree:
  ```
  bootz 0x80200000 - 0x80F80000
  ```

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

```
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(1,0)
```

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

Reset the board once more (**User Button**!) and wait until the `bootcmd` has run its course.

```
run bootcmd
```

Now the boot will proceed without a lot of user input, since everything is handled by the new environment.

If you NFS is setup properly, the kernel may stop because of the below issue:

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

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

Finally, the boot will terminate with the following error message:

```
Kernel panic - not syncing: No init found.  Try passing init= option to kernel. See Linu
```

The reason is that the root file system is empty, and the kernel does not find the `init` file.

We will handle this in forthcoming labs.

Automating the boot

You can automate all this every time the board is booted or reset.

U-Boot contains a precompiled environment, which will attempt to load the kernel from an SD-Card.

Remove the micro-SD card and connect it to your host.

Verify the `uEnv.txt`.

```
ipaddr=192.168.0.100
serverip=192.168.0.1
loadaddr=0x80200000
fdtaddr=0x80F80000
IMAGE=rootfs
console=ttyO0,115200n8
nfsopts=nolock
netargs=setenv bootargs console=${console} root=/dev/nfs \
  nfsroot=${serverip}:/tftpboot/${IMAGE},${nfsopts} rw ip=${ipaddr}
tftp_kernel=tftp ${loadaddr} zImage
tftp_dtb=tftp ${fdtaddr} am335x-boneblack.dtb
bootkernel=bootz ${loadaddr} - ${fdtaddr}
uenvcmd=run tftp_kernel tftp_dtb netargs bootkernel
mmcboot=echo MMC boot disabled
nandboot=echo NAND boot disabled
```

Note that the netargs line should be a single line.

Remember to run `sync` before removing the card.

Reinsert the card in the Beaglebone Black and type

```
reset
```

Do not press the **return** key, to let the `bootcmd` execute.

After loading the `uEnv.txt` environment, U-Boot is configured to test for the existence of the `uenvcmd` variable, and this will be run, if it exists.

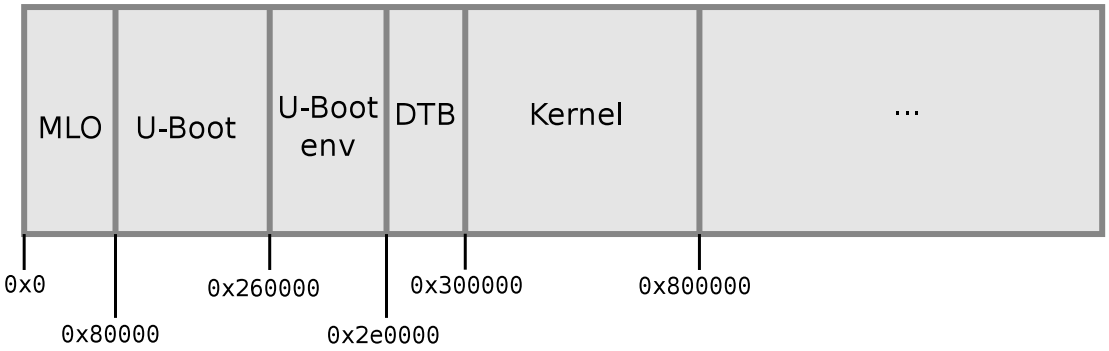
Since `uenvcmd` now is defined, the kernel will start to load, but will again abort due to no `init`.

Flashing the kernel and DTB in NAND flash (Skip))

On a system with NAND Flash, you can program the kernel into the flash after downloading with TFTP. The Beaglebone Black does not have NAND Flash so this section is for reference only. Read through when you have time, but do not perform the commands.

In order to let the kernel boot on the board autonomously we can, on a board with NAND flash, flash the kernel image and DTB in the NAND flash. See the bootloader lab for details about U-boot's `nand` command.

After storing the first stage bootloader, U-boot and its environment variables, we will keep special areas in NAND flash for the DTB and Linux kernel images:



So, let's start by erasing the corresponding 128 KiB of NAND flash for the DTB:

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

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

```
nand erase 0x300000 0x500000
```

Then, copy the DTB and kernel binaries from TFTP into memory, using the same addresses as before.

Then, flash the DTB and kernel binaries:

```
nand write 0x81000000 0x2e0000 0x20000
           (RAM addr) (NAND offset) (size)
nand write 0x80000000 0x300000 0x500000
```

Power your board off and on, to clear RAM contents. We should now be able to load the DTB and kernel image from NAND and boot with:

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

NAND boot `nboot` copies the kernel to RAM, using the `uImage` headers to know how many bytes to copy. You could have used `nand read 0x80000000 0x300000 0x500000`, but you would have copied more bytes than the actual size of your kernel. [11]. Modern Linux uses `zImage` which is not supported by `nboot`, so the command is not appropriate for newer kernels.

[11] `nboot` can save a lot of boot time, as it avoids having to copy a pessimistic number of bytes from flash to RAM

When TFTP downloads a file, it will store the size of the file in the 'filesize' environment variable. The value can be stored in another variable, I.E: 'kernel_size'

```
tftp 80000000 zImage;
setenv kernel_size $filesize
tftp 81000000 am335x-boneblack.dtb;
setenv dtb_size $filesize
```

This can be used to retreive the files without redundant reads:

```
nand read 0x80000000 0x300000 $kernel_size
nand read 0x81000000 0x2e0000 $dtb_size
bootz 80000000 - 81000000
```

Note that U-boot is not always configured with `nboot` support.

Write a U-Boot script that automates the DTB + kernel download and flashing procedure. Finally, adjust `bootcmd` so that the 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.

Tiny embedded system with BusyBox

Objective: making a tiny yet full featured embedded system

After this lab, you will:

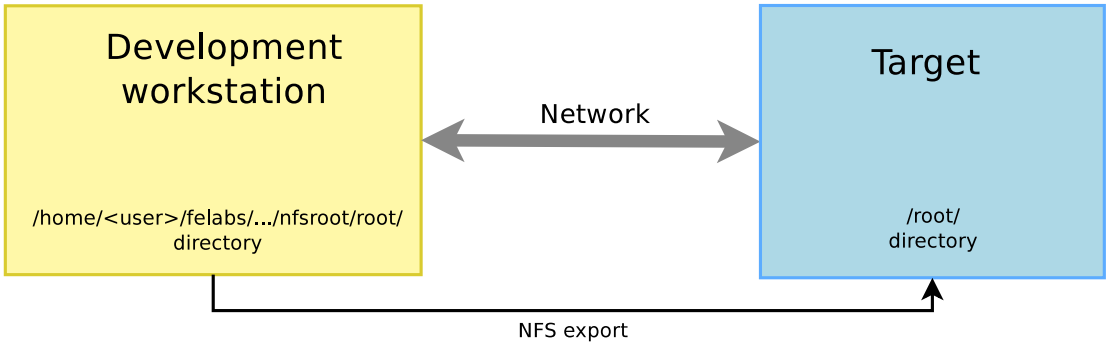
- be able to configure and build a Linux kernel that boots on a directory on your workstation, shared through the network by NFS.
- be able to create and configure a minimalistic root filesystem from scratch (ex nihilo, out of nothing, entirely hand made...) for the Beaglebone Black board
- understand 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 developing a root filesystem for a device, the 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 `tinysystem` subdirectory of `~/felabs/sysdev` .

Kernel configuration

We will re-use the kernel sources from our previous lab, in the `kernel` subdirectory of `~/felabs/sysdev` .

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

Make sure your NFS directory is setup.

Booting the system - Skip if you did this in the kernel lab

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.

If you did not setup your `uEnv.txt` file in the SD-Card FAT partition, do so now:

Add the following U-Boot commands to your uEnv.txt to do so, **netargs in just 1 line**:

```
ipaddr=192.168.0.100
serverip=192.168.0.1
loadaddr=0x80200000
fdtaddr=0x80F80000
IMAGE=rootfs
console=ttyO0,115200n8
nfsopts=nolock
netargs=setenv bootargs console=${console} root=/dev/nfs \
  nfsroot=${serverip}:/tftpboot/${IMAGE},${nfsopts} rw ip=${ipaddr}
tftp_kernel=tftp ${loadaddr} zImage
tftp_dtb=tftp ${fdtaddr} am335x-boneblack.dtb
bootkernel=bootz ${loadaddr} - ${fdtaddr}
uenvcmd=run tftp_kernel tftp_dtb netargs bootkernel
mmcboot=echo MMC boot disabled
nandboot=echo NAND boot disabled
```

Of course, you need to adapt the IP addresses to your exact network setup.

You will later need to make changes to the `bootargs` value.

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

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

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

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

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

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

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

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


Getting the Busybox source code

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

Goto the `~/felabs/sysdev` directory and then to the `tinysystem` subdirectory.

Download the latest BusyBox 1.22.1 release from `http://www.busybox.net/downloads`.

You should have run `make prepare` initially in `~/felabs/sysdev` .

If not,install the required packages: [12]

```
sudo apt-get install libglade2-dev libglib2.0-dev libgtk2.0-dev
```

Unpacking and Patching the Busybox source code

You will build Busybox in several configurations, each in a separate subdirectory. Everytime you build a new configuration, you need to follow this procedure.

- Go to the subdirectory.
- Unpack the busybox sources
- Initialize an empty git inside the busybox source directory.
- Apply the patches found in the `patches` subdirectory of tinysystem, in alphabetical order.
- Make sure that you do not have any modified or new files if you do `git status`.

Configuring Busybox with static libraries

Ensure that you have the toolchain set-up, and enter the `static` subdirectory.

Unpack and patch the busybox source.

You can get the default Busybox configuration (`.config`) by:

```
make defconfig
```

This time, Busybox will be built using a predefined configuration file located in the `static` sub-directory. You should use `busybox-1.22.1.config`. Copy this file to the Busybox configuration file `.config` in the Busybox sources.

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

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

Run `make gconfig` to get the configuration window.

Look around in the configuration, and see what options you have.

You will need to set LDFLAGS before the build.

It can be done in the shell, or it can be set in the configuration.

[12] If **make gconfig** fails later due to lack of GTK+, you can also try **make menuconfig**

```
export LDFLAGS="--static"
```

Specify the installation directory for BusyBox [13]. It should be the path to your `nfsroot` directory.

Alternatively, you can set `_install` directory as a dynamic link to `/tftpboot/rootfs`

Exit the configuration, saving changes.

Building Busybox

Source the `toolchain.sh` script and start the build by typing `make`

Installing Busybox

For this lab, we will create our NFS image in `/tftpboot/busybox.static`

Create the directory and make sure it can be NFS mounted. Don't forget to restart the NFS server.

If you just type `make install` all files will be installed with You as the owner, but the proper owner for these files is `root`.

If you `sudo make install`, the files will be installed with the proper owner, but you will also clear the environment, which means that `ARCH` is empty, and defaults to the host machine `x86_64`. This means that all files compiled for ARM, will be recompiled, and you get a busybox image for the host, instead of for the **Beaglebone Black** .

The proper way, is to enter superuser modes, setup the environment, and then install

```
sudo su
source toolchain.sh
ln -s /tftpboot/busybox.static _install
make install
```

Verify that you have the right architecture, by entering the `/tftpboot/busybox.static/bin` directory and run `arm-poky-linux-gnueabi-readelf -a busybox`

Also create a `dev` directory in `nfsroot` if it does not exist.

Copy your kernel and device-tree file to `boot` directory of your filesystem.

'sudo chmod -s busybox

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

Setting up U-Boot for testing Busybox

Copy the `uSetup.txt` from the subdirectory to the SD-card.

```
ipaddr=192.168.0.100
serverip=192.168.0.1
loadaddr=0x80200000
fdtaddr=0x88000000

bootdir=/boot
bootfile=zImage

console=ttyO0,115200n8
nfsopts=nolock
optargs=video=HDMI-A-1

setpath=setenv rootpath /tftpboot/${image}
settftp=setenv rootpath ${image}
setup_ip=setenv ip ${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}::off

tftp_kernel=run settftp ; tftp ${loadaddr} ${rootpath}/${bootdir}/${bootfile}
tftp_dtb=run settftp    ; tftp ${fdtaddr}  ${rootpath}/${bootdir}/${fdtfile}

mmcargs=setenv bootargs console=${console} ${optargs} root=${mmcroot}   \
  rootfstype=${mmcrootfstype} rootwait
mmcrootfstype=squashfs
nandboot=echo NAND boot disabled

bootzImage=if test "${bootfile}" = "zImage";then bootz ${loadaddr} - ${fdtaddr}; fi
bootuImage=if test "${bootfile}" = "uImage";then bootm ${loadaddr} - ${fdtaddr}; fi
bootkernel=run bootzImage ; run bootuImage

netargs=run setpath; run setup_ip ; setenv bootargs console=${console} \
  ${optargs} root=/dev/nfs nfsroot=${serverip}:${rootpath},${nfsopts} rw ip=${ip}
netboot=echo Booting from network ...; run findfdt; setenv autoload no; \
  run tftp_kernel ; run tftp_dtb ; run netargs; bootm ${loadaddr} - ${fdtaddr}

# image=core-image-minimal
image=busybox.static
br=setenv         image buildroot          ; setenv bootfile zImage ; run bootcmd
dynamic=setenv    image busybox.dynamic    ; setenv bootfile zImage ; run bootcmd
httpd=setenv      image busybox.httpd      ; setenv bootfile zImage ; run bootcmd
static=setenv     image busybox.static     ; setenv bootfile zImage ; run bootcmd

thirdparty=setenv image thirdparty         ; setenv bootfile zImage ; run bootcmd

minimal=setenv    image core-image-minimal ; setenv bootfile uImage ; run bootcmd
base=setenv       image core-image-base    ; setenv bootfile uImage ; run bootcmd
cato=setenv       image core-image-cato    ; setenv bootfile uImage ; run bootcmd

bootcmd=run findfdt ; run tftp_kernel tftp_dtb netargs bootkernel
```


Testing Busybox

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

Virtual filesystems

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

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

Now mount the `proc` virtual filesystem.

```
mount -t proc nodev /proc
```

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 [14] and a `/etc/init.d/rcS` startup script declared in `/etc/inittab`. In this startup script, mount the `/proc` and `/sys` filesystems.

Reboot and check the contents. They should not be empty.

Any issue after doing this?

Switching to shared libraries

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

```
${CC}          data/hello.c -o hello
${CC} --static data/hello.c -o Hello
```

Copy the files to the `/usr/bin` directory in the `nfsroot`

Try run `Hello` on the **Beaglebone Black** . This should work OK.

Then try to run `hello`

You will first encounter a `not found` error caused by the absence of the `ld-linux-armhf.so.3` executable, which is the dynamic linker required to execute any program compiled with shared libraries. Using the find command look for this file in the toolchain install directory.

This turns out to be a link to `ld-2.19.so`, so copy `ld-2.19.so` to the `lib/` directory of the target.

[14] Do not use getty as a shell, use /bin/sh

Setup the link:

```
ln -s ld-2.19.so /lib/ld-linux-armhf.so.3
```

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. Setup any needed links.

```
ln -s libc-2.19.so libc.so.6
```

If you get a permission denied error, you may have to set the execute bit of the libraries.

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

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

Configuring Busybox with shared libraries

Create the `/tftpboot/busybox.httpd` directory and export it via NFS.

Restart the `nfs-kernel-server`, so NFS still works.

Go to the `httpd` subdirectory.

Use the `busybox-1.22.1.config` in `httpd` as your config.

Build Busybox with shared libraries, and install it on the target filesystem.

Copy non busybox parts from the previous build. Dont forget the kernel files.

Create the `dynamic` textfile in the toplevel.

Time to boot the system. How can you make sure that you are booting from the right NFS export.

What options do you have?

Make sure that the system still boots and see how much smaller the `busybox` executable got.

```
ls -l /
```

You should see the `dynamic` file.

Implement a web interface for your device

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

```
rsync -av <src> <dst>
```

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

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

It will automatically background itself.

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

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

How much RAM does your system need?

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

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 **Beaglebone Black** will be booted with this MMC card, without using NFS anymore.

Setup

Throughout this lab, we will continue to use the root filesystem we have created in the `/tftpboot/busybox.httpd` directory, which we will progressively adapt to use block filesystems.

Filesystem support in the kernel

Your kernel should already be compiled with support for SquashFS and ext3/4. If not, reconfigure the kernel and recompile and copy to the `boot` directory of `/tftpboot/busybox.httpd`. Also install the kernel modules.

Boot your board with this new kernel and on the NFS filesystem you used in this previous lab.[15]

Add partitions to the MMC card

If you followed the setup of the SD-Card, it should be ready for use. Otherwise go back and setup the SD-card.

Using `cfdisk` [16] (or `cfdisk /dev/mmcblk0x`).

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

[16] When you have initialized a partition and want to add more, you don't have to specify headers and sectors again. Just run `cfdisk /dev/sdx`

On top of the boot partition, we will need:

- One partition that will be used for the root filesystem. Due to the geometry of the device, the minimum partition size must be larger than 8 MB, but if we add kernel modules, the size will be much larger. We will use 2048 MB. Keep the `Linux` type for the partition.
- One partition, that will be used for the data filesystem. Here also, keep the `Linux` type for the partition. This should be 512 MB.

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

Connect the MMC disk to your board and boot using NFS. You should see the MMC partitions in `/proc/partitions`.

On the board, create the `/media` directory, and mount the third partition labeled `data`.

```
mount -t ext4 /dev/mmcblk0p3 /media
```

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.

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

Test it by starting `httpd` and check with a web-browser.

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

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

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

Adding a tmpfs partition for log files

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

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

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

Reboot your system and check that it works as expected.

Making a SquashFS image

We are going to store the root filesystem in a SquashFS filesystem in the second partition of the MMC disk.

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

Before creating the filesystem, you should create the `boot` directory, in `/tftpboot/busybox.httpd` and copy the kernel and device tree file to this directory.

```
cd /tftpboot
mksquashfs -all-root -noappend busybox.httpd rootfs.sqfs
```

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

Be **very careful**, that you do not destroy a hard disk.

```
sudo dd if=rootfs.sqfs of=/dev/sdb2
```

Booting on the SquashFS partition

In the U-boot shell, configure the kernel command line to use the second partition of the MMC disk (`/dev/mmcblk0p2`) as the root file system. (YOU can also modify the `uSetup.txt` file.

Remeber that there is a predefined environment which you can use.

Also add the `rootwait` boot argument, to wait for the MMC disk to be properly initialized before trying to mount the root filesystem. Since the MMC cards are detected asynchronously by the kernel, the kernel might try to mount the root filesystem too early without `rootwait`.

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

Store the kernel image and DTB on the MMC card

Finally, copy the `zImage` kernel image and DTB to the SD-Card. What is the easiset? You can put it in the first partition of the SD card (the partition called `boot`), or you can keep it in the root filesystem. Edit the `uSetup.txt` to adjust the `bootcmd` of U-Boot so that it loads the kernel and DTB from the SD card instead of loading them through the network.

Booting the easy way

Reset the board, stop the boot, and try `run mmcboot`.

If you set up the correct parameters in `uSetup.txt`, it should work like a charm.

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.

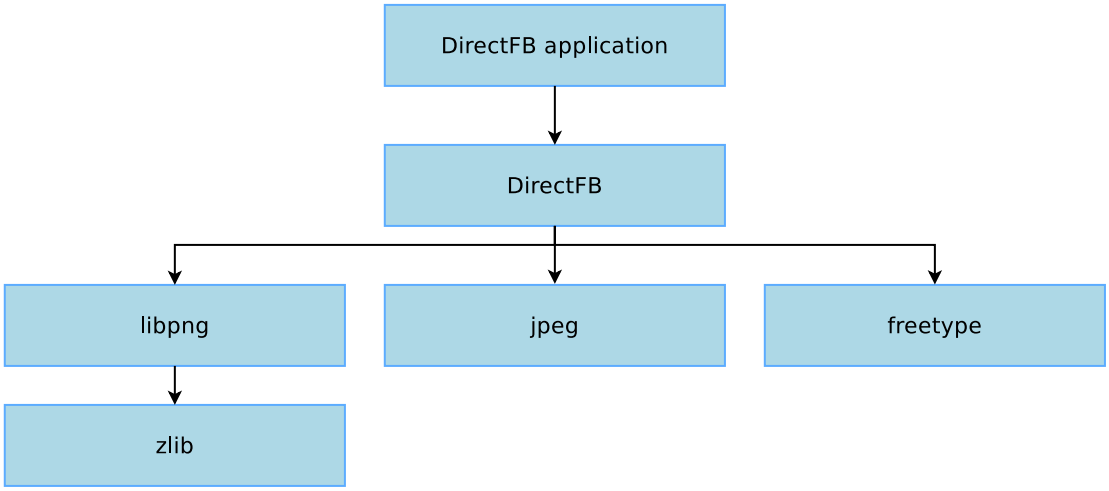
We'll see that manually re-using existing libraries is quite tedious, so that more automated procedures are necessary to make it easier. However, learning how to perform these operations manually will significantly help you when you'll face issues with more automated tools.

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 [17], 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.

In `$HOME/felabs/sysdev/thirdparty`, create the `staging` directories and create a symbolic link from `target` to `/tftpboot/core-image-minimal`

For the target, we need a basic system with BusyBox, device nodes and initialization scripts. Since the kernel build we have done, has an issue with the framebuffer we are going to use a better tested Yocto file system.

Unpack `ftp://ftp.emagii.com/pub/training/yocto/core-image-minimal-beaglebone.tar.bz2` in `/tftpboot`.

```
cd /tftpboot
wget ftp://www.emagii.com/pub/training/yocto/core-image-minimal-beaglebone.tar.bz2
mkdir -p core-image-minimal
cd core-image-minimal
sudo tar --numeric-owner -jxvf ../core-image-minimal-beaglebone.tar.bz2
```

It would be nice to track every change we do to the root filesystem, so lets initialize a git repo inside the filesystem.

```
cd /tftpboot/core-image-minimal
git init
git add .
git commit -m "Initial Commit" -s
```

ls /tftp We will also store the kernel and device tree file inside the filesystem in the `/boot` directory.

```
mkdir -p boot
cd boot
sudo wget ftp://ftp.emagii.com/pub/training/yocto/uImage
sudo wget ftp://ftp.emagii.com/pub/training/yocto/am335x-boneblack.dtb
```

Check in the new files using `git`

To make it easy to use the directory, we will make a symbolic link to this directory from the third party directory.

Go back to `$HOME/felabs/sysdev/thirdparty` and create a link.

```
ln -s /tftpboot/core-image-minimal target
```

[17] You may run into problems with write permission, when you are stripping in the target directory, so you may want to use a separate terminal window, running in supervisor more with the path setup to the toolchain

Testing

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

```
/tftpboot/core-image-minimal *(rw,sync,no_subtree_check,no_root_squash)
```

And restart your NFS server.

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

You need a little different `uSetup.txt`. There should be a working `uSetup.txt` in the `$HOME/felabs/sysdev/thirdparty/data` directory. [18]

The system should boot and give you a login prompt.

Use `root`. Password will not be needed.

If the system boots OK, a few files have been created. Check in the files using git on your host. (YOu may have to change the permissions on some files).

You can use the commands `poweroff` or `reboot` to stop the system.

[18] If you have built a Yocto you will also get **MLO** and **u-boot.img**. They do not contain the patch to use **uSetup.txt**

uSetup.txt

Note: Lines split, needs to be on a single line in the file.

```
ipaddr=192.168.0.100
serverip=192.168.0.1
loadaddr=0x80200000
fdtaddr=0x88000000
HOME=/home/ulf
bootdir=/boot
bootfile=uImage

console=ttyO0,115200n8
nfsopts=nolock
optargs=video=HDMI-A-1

setpath=setenv rootpath /tftpboot/${image}
setup_ip=setenv ip ${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}::off

tftp_kernel=run setpath ; tftp ${loadaddr} ${rootpath}/${bootdir}/${bootfile}
tftp_dtb=run setpath    ; tftp ${fdtaddr}  ${rootpath}/${bootdir}/${fdtfile}

mmcargs=setenv bootargs console=${console} ${optargs} root=${mmcroot} \
  rootfstype=${mmcrootfstype} rootwait
mmcrootfstype=squashfs
nandboot=echo NAND boot disabled

bootzImage=if test "${bootfile}" = "zImage"; then \
  bootz ${loadaddr} - ${fdtaddr}; fi
bootuImage=if test "${bootfile}" = "uImage"; then \
  bootm ${loadaddr} - ${fdtaddr}; fi
bootkernel=run bootzImage ; run bootuImage

netargs=setenv bootargs console=${console} ${optargs} \
  root=/dev/nfs nfsroot=${serverip}:/tftpboot/${image},${nfsopts} rw ip=${ipaddr}
netargs=run setpath     ; run setup_ip ; setenv bootargs console=${console} \
  ${optargs} root=/dev/nfs nfsroot=${serverip}:${rootpath},${nfsopts} rw ip=${ip}
netboot=echo Booting from network ...; run findfdt; setenv autoload no; \
  run tftp_kernel ; run tftp_dtb ; run netargs; bootm ${loadaddr} - ${fdtaddr}

image=core-image-minimal
br=setenv         image buildroot          ; setenv bootfile zImage ; run bootcmd
dynamic=setenv    image busybox.dynamic    ; setenv bootfile zImage ; run bootcmd
static=setenv     image busybox.static     ; setenv bootfile zImage ; run bootcmd
thirdparty=setenv image thirdparty         ; setenv bootfile zImage ; run bootcmd
minimal=setenv    image core-image-minimal ; setenv bootfile uImage ; run bootcmd
base=setenv       image core-image-base    ; setenv bootfile uImage ; run bootcmd
cato=setenv       image core-image-cato    ; setenv bootfile uImage ; run bootcmd

bootcmd=run findfdt ; run tftp_kernel tftp_dtb netargs bootkernel
```

zlib

`Zlib` is a compression/decompression library available at http://www.zlib.net/. Download version 1.2.8, and extract it in `$HOME/felabs/sysdev/thirdparty/`.

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

Let's try to configure and build zlib:

```
./configure
make
```

You can see that the files are getting compiled with gcc, which generates code for x86 and not for the target platform. This is obviously not what we want, so we tell the configure script to use the ARM cross-compiler by setting **CC**. This is done by our toolchain script so we need to run this before `./configure`.

Enter the `zlib` source directory.

```
source ../../toolchain.sh
./configure
```

```
# 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:

```
${CROSS_COMPILE}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.1.2.8`, it has been generated by the following command line:

```
${CROSS_COMPILE}gcc -shared -Wl,-soname,libz.so.1 -o libz.so.1.2.8 \
              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.1.2.8 libz.so
ln -s libz.so.1.2.8 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:

```
${CROSS_COMPILE}readelf -d libz.so.1.2.8
```

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 1.2.8, 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.1.2.8` 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:

```
./configure --prefix=/usr
make
```

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

Do not confuse the *prefix* (where the application or library will be running on the target system) from the location where the application or library will be installed on your host while building the root filesystem. For example, zlib will be installed in `$HOME/felabs/sysdev/thirdparty/target/usr/lib/` because this is the directory where we are building the root filesystem, but once our target system will be running, it will see zlib in `/usr/lib`. The prefix corresponds to the path in the target system and **never** on the host. So, one should **never** pass a prefix like `$HOME/felabs/sysdev/thirdparty/target/usr`, otherwise at runtime, the application or library may look for files inside this directory on the target system, which obviously doesn't exist! By default, most build systems will install the application or library in the given prefix (`/usr` or `/usr/local`), but with most build systems (including *autotools*), the installation prefix can be overriden, and be different from the configuration prefix.

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

```
make DESTDIR=/home/ulf/felabs/sysdev/thirdparty/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.1.2.8` are needed, since `libz.so.1` is the *SONAME* of the library and `libz.so.1.2.8` is the real binary:
   ```
   cp -a libz.so.1* ../target/usr/lib
   ```
3. Strip the library:
   ```
   ${CROSS_COMPILE}strip ../target/usr/lib/libz.so.1.2.8
   ```

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:

```
source ../../toolchain.sh
./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:

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

This time, the `./configure` completes, but there is a problem.

The configure script tries to compile an application against *zlib*. *libpng* uses the *zlib* library, so the `configure` script wants to make sure this library is already installed. Unfortunately, the `ld` linker find the library in the toolchain, and not the one we compiled. 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:

```
export LDFLAGS=-L$HOME/felabs/sysdev/thirdparty/staging/usr/lib
./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`:

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

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

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

Let's use it:

```
export CPPFLAGS=-I$HOME/felabs/sysdev/thirdparty/staging/usr/include
./configure --host=arm-linux --prefix=/usr
```

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

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

```
make DESTDIR=/tmp/libpng/ install
```

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

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

```
./usr/lib/libpng.la                     -> libpng14.la
./usr/lib/libpng14.a
./usr/lib/libpng14.la
./usr/lib/libpng14.so                   -> libpng14.so.14.3.0
./usr/lib/libpng14.so.14                -> libpng14.so.14.3.0
./usr/lib/libpng14.so.14.3.0
```

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

So, we have:

- The library, with many symbolic links
  - `libpng14.so.14.3.0`, the binary of the current version of library
  - `libpng14.so.14`, a symbolic link to `libpng14.so.14.3.0`, so that applications using `libpng14.so.14` as the *SONAME* of the library will find nit and use the current version
  - `libpng14.so` is a symbolic link to `libpng14.so.14.3.0`. So it points to the current version of the library, so that new applications linked with `-lpng14` will use the current version of the library `libpng.so` is a symbolic link to libpng14.so. So applications linked with `-lpng` will be linked with the current version of the library (and not the obsolete one since we don't want anymore to link applications against the obsolete version!)
  - `libpng14.a` is a static version of the library
  - `libpng.a` is a symbolic link to `libpng14.a`, so that applications statically linked with `libpng.a` will in fact use the current version of the library
  - `libpng14.la` is a configuration file generated by *libtool* which gives configuration details for the library. It will be used to compile applications and libraries that rely on libpng.
  - `libpng.la` is a symbolic link to `libpng14.la`: we want to use the current version for new applications, once again.
- The *pkg-config* files, in `/usr/lib/pkgconfig/`. These configuration files are used by the pkg-config tool that we will cover later. They describe how to link new applications against the library.
- The manual pages in `/usr/share/man/`, explaining how to use the library.
- The header files, in `/usr/include/`, needed to compile new applications or libraries against *libpng*. They define the interface to *libpng*. There are symbolic links so that one can choose between the following solutions:
  - Use `#include <png.h>` in the source code and compile with the default compiler flags
  - Use `#include <png.h>` in the source code and compile with `-I/usr/include/libpng14`

  - Use `#include <libpng14/png.h>` in the source and compile with the default compiler flags

- The `/usr/bin/libpng14-config` tool and its symbolic link `/usr/bin/libpng-config`. This tool is a small shell script that gives configuration information about the libraries, needed to know how to compile applications/libraries against libpng. This mechanism based on shell scripts is now being superseded by *pkg-config*, but as old applications or libraries may rely on it, it is kept for compatibility.

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

```
make DESTDIR=/home/ulf/felabs/sysdev/thirdparty/staging/ install
```

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

```
cd ..
sudo cp -a staging/usr/lib/libpng14.so.* target/usr/lib
${CROSS_COMPILE}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:

To save time in the future, we will use a modified `toolchain.sh`. Copy the file to the `thirdparty` directory and add the following lines with the corretc user of course.

```
export LDFLAGS=-L/home/ulf/felabs/sysdev/thirdparty/staging/usr/lib
export CPPFLAGS=-I/home/ulf/felabs/sysdev/thirdparty/staging/usr/include
```

You do not want to use the `$HOME` variable, since sometimes you run the toolchain.sh script from the `root` user, and then they would point at the wrong location. You do need to replace `ulf` with your own username.

```
. ../toolchain.sh
./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:

```
sudo make DESTDIR=/home/ulf/felabs/sysdev/thirdparty/staging/ install
```

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

```
cd ..
sudo cp -a staging/usr/lib/libjpeg.so.8* target/usr/lib/
${CROSS_COMPILE}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:

```
source ../toolchain.sh
./configure --host=arm-linux --prefix=/usr
```

Then, compile the library:

```
make
```

Install it in the *staging* space:

```
sudo make DESTDIR=/home/ulf/felabs/sysdev/thirdparty/staging/ install
```

Obviously change the user to your own username.

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

```
cd ..
sudo cp -a staging/usr/lib/libfreetype.so.6* target/usr/lib/
${CROSS_COMPILE}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:

```
source ../toolchain.sh
./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/opt/poky/sysroot/<...> -lpng12
[...]
FreeType2         yes  -I/opt/poky/sysroot/<...> -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 in our SDK 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 the `core-image-sato` SDK, then the configure script of DirectFB found the libpng and FreeType libraries there!

If it wasn't part of the SDK, then *pkg-config* would have found the host libraries.

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 on the host, nor in the SDK*pkg-config*, then the `/usr/lib/pkgconfig/freetype2.pc` file wouldn't exist, and the configure script of DirectFB would have said something like *Sorry, can't find FreeType*.

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

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

```
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include
[...]
Libs: -L${libdir} -lfreetype
Cflags: -I${includedir}/freetype2 -I${includedir}
```

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

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

Add the following lines to your thirdparty `toolchain.sh`, modifying the user.

```
export PKG_CONFIG_PATH=/home/ulf/felabs/sysdev/thirdparty/staging/usr/lib/pkgconfig
export PKG_CONFIG_SYSROOT_DIR=/home/ulf/felabs/sysdev/thirdparty/staging
```

The source the new configuration and configure:

```
. ../toolchain.sh
./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
```

If we try building DirectFB now, it will complete the build, but there may be one remaining problem. If configure for some reason finds X11, we have not added that to our image.

If DirectFB has enabled X11, it fails during the build, complaining that `X11/Xlib.h` and other related header files cannot be found. In fact, if you look back the `./configure` script output, you in that case see:

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

Because X11 was installed in the serach path, 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]
```

If X11 is enabled, 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/ulf/felabs/sysdev/thirdparty/staging/ install
```

And so the installation in the *target* space:

- First, the libraries:

```
cd ..
sudo su
source ./toolchain.sh
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
${CROSS_COMPILE}strip target/usr/lib/libdirect-1.4.so.5.0.0
${CROSS_COMPILE}strip target/usr/lib/libdirectfb-1.4.so.5.0.0
${CROSS_COMPILE}strip target/usr/lib/libfusion-1.4.so.5.0.0
exit
```

- 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:

```
sudo su
source toolchain.sh
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 ${CROSS_COMPILE}strip {} \;
exit
```

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:

```
source toolchain.sh
cd DirectFB-examples-1.2.0
./configure --host=arm-linux --prefix=/usr
```

Then, compile it with `make`. 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):

```
sudo su
source ../toolchain.sh
make STRIP=${CROSS_COMPILE}strip \
     DESTDIR=/home/ulf/felabs/sysdev/thirdparty/target/ install-strip
```

Final setup

If you try to run the DirectFB examples now, they will fail to run, because of missing libraries. The `core-image-minimal` filesystem includes some needed libraries like `libpthread` and `libdl`.

`libpthread` is used to implement threads.

`libdl`, is used to dynamically load libraries during application execution.

`libsysfs` and `libgcc_s` are missing.

Add the following line to the end of the thirdparty `toolchain.sh`

```
export TOOLCHAIN_SYSROOT=$(${CROSS_COMPILE}gcc -print-sysroot)
```

So let's add the missing libraries to the target:

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

Now, the application should no longer complain about missing libraries.

Other problems that could occur are missing device files.

In a busybox barebone, you would need `/dev/fb0` which can be created by:

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

`Yocto` which is used by the `core-image-minimal` used `udev` which dynamically creates the device files at run time, and is a more modern approach.

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

Check that you do not get any error messages.

Without a screen, you will not be able to do a lot, but you may be able to borrow one of the few screens with HDMI input that are available.

If you want to try this at home, you will need a microHDMI to HDMI cable.

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

Setup

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

Get tarballs

If you have a slow internet connection, and have a DVD with the needed tarballs you can copy the `buildroot/download` directory on the DVD to `$HOME/felabs/sysdev/buildroot/download`

Get Buildroot and explore the source code

The official Buildroot website is available at `http://buildroot.org/`. Download the latest stable 2014.02.x version which we have tested for this lab. Uncompress the tarball and go inside the Buildroot source directory.

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

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

- `toolchain` contains the Makefiles, patches and configuration items to generate the cross-compiling toolchain;

Set the download directory

If you got the tarballs from a DVD, link the buildroot download directory `${TOPDIR}/dl` to your download directory.

```
ln -s ../download dl
```

Configure Buildroot

In our case, we would like to:

- Having Buildroot generate a toolchain for us;
- Generate an embedded Linux system for ARM;
- Integrate Busybox in our embedded Linux system;
- Integrate the target filesystem into both an ext4 filesystem image and a tarball

To run the configuration utility of Buildroot, simply run:

```
make beaglebone_defconfig
```

Then we want to customize the build.

```
make menuconfig
```

Set the following options:

- `Toolchain`
  - enable `Toolchain has large file support?`,
  - enable `Toolchain has RPC support?`
  - enable `WCHAR support?`
  - enable `Thread Library Debugging`
  - enable `Toolchain has C++ support?`.
  - enable `Build cross-gdb for the host`.
- `Target packages`
  - Keep `BusyBox` (default version) and keep the Busybox configuration proposed by Buildroot;
  - in `libraries->networking`
    - Select `libcurl` and `curl` binary
  - in `libraries->others`
    - Select `boost` (Enable all options)
- `Target Packages`
  - Select `Debugging, profiling and benchmark`
    - Select `gdb->gdbserver`
    - Select `strace`
- `Filesystem images`
  - Select `ext4 root filesystem`
  - Select `tar the root filesystem with bzip2`

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

Generate the embedded Linux system

Just run:

```
make
```

Buildroot will download, extract, configure, compile and install each component of the embedded system.

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

- `build`, is the directory in which each component built by Buildroot is extract, and where the build actually takes place
- `host`, is the directory where Buildroot installs some components for the host. As Buildroot doesn't want to depend on too many things installed in the developer machines, it installs some tools needed to compile the packages for the target. In our case it installed pkg-config (since the version of the host may be ancient) and tools to generate the root filesystem image (genext2fs, makedevs, fakeroot)
- `images`, which contains the final images produced by Buildroot. In the normal case it's just an ext2 filesystem image and a tarball of the filesystem, but depending on the Buildroot configuration, there could also be a kernel image or a bootloader image. This is where we find `rootfs.tar` and `rootfs.ext2`, which are respectively the tarball and the ext2 image of the generated root filesystem.
- `staging`, which contains the build space of the target system. All the target libraries, with headers, documentation. It also contains the system headers and the C library, which in our case have been copied from the cross-compiling toolchain.
- `target`, is the target root filesystem. All applications and libraries, usually stripped, are installed in this directory. However, it cannot be used directly as the root filesystem, as all the device files are missing: it is not possible to create them without being root, and Buildroot has a policy of not running anything as root.
- `toolchain`, is the location where the toolchain is built.

Run the generated system

Create a directory `/tftpboot/buildroot`, where you extract the tarball.

```
mkdir -p /tftpboot/buildroot
cd       /tftpboot/buildroot
sudo tar --numeric-owner -jxvf rootfs.tar.bz2
```

Make sure the filesystem is exported through NFS, including restarting the NFS server.

Update the `uSetup.txt` file on your SD-Card to allow boot from buildroot

Softlink `${HOME}/rootfs` to this place and make sure you restart the NFS server.

Copy the `zImage` and the device tree file to `/tftpboot/buildroot/boot` directory.

Boot using NFS.

Use the `root` username, no password is required.

Application development

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

Setup

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

Compile your own application

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

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

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

Now, our application. In the lab directory the file `data/app.c` contains a very simple DirectFB application that displays the `data/background.png` image for five seconds. We will compile and integrate this simple application to our Linux system.

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

Let's add this directory to our PATH:

```
export PATH=$HOME/felabs/sysdev/buildroot/buildroot-XXXX.YY/output/host/usr/bin:$PATH
```

Let's try to compile the application:

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

It complains that it cannot find the directfb.h header. This is normal, since we didn't tell the compiler where to find it. So let's use `pkg-config` to query the *pkg-config* database about the location of the header files and the list of libraries needed to build an application against DirectFB:

```
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!

Remote application debugging

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

Setup

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

Debugging setup

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

Setting up gdbserver and strace

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

Enabling job control

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

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

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

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

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

Replace

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

which implied the use of the system console device by

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

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

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

Using strace

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

With your cross-compiling toolchain, compile the `data/vista-emulator.c` program, strip it with `arm-linux-strip`, and copy the resulting binary to the `/root` directory of the root filesystem (you might need to create this directory if it doesn't exist yet).

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

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

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

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

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

Using gdbserver

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

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

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

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

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

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

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

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

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

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

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

And tell gdb to connect to the remote system:

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


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

```
(gdb) continue
```

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

```
(gdb) backtrace
```

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

What to remember

During this lab, we learned that...

- Compiling an application for the target system is very similar to compiling an application for the host, except that the cross-compilation introduces a few complexities when libraries are used.
- It's easy to study the behavior of programs and diagnose issues without even having the source code, thanks to strace.
- You can leave a small `gdbserver` program (300 KB) on your target that allows to debug target applications, using a standard GDB debugger on the development host.
- It is fine to strip applications and binaries on the target machine, as long as the programs and libraries with debugging symbols are available on the development host.

Real-time - Timers and scheduling latency

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

After this lab, you will:

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

Setup

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

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

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

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

Install the `netcat` package.

Root filesystem

Create an `nfsroot` directory.

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

Let's build this with Buildroot.

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

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

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

- `Target Architecture Variant`: `cortex-a8`
- In `Toolchain`:
  - `Toolchain type`: `External toolchain`
  - `Toolchain`: `Sourcery CodeBench ARM 2012.03`
- In `System configuration`:
  - `/dev management`: `Dynamic using devtmpfs only`
  - `Port to run a getty (login prompt) on`: `ttyO2`
- In `Package Selection for the target`:
  - Enable `Show packages that are also provided by busybox`. We need this to build the standard `netcat` command, not provided in the default BusyBox configuration.
  - In `Debugging, profiling and benchmark`, enable `rt-tests`. This will be a few applications to test real-time latency.
  - In `Networking applications`, enable `netcat`
  - In `Real-Time`, enable `Xenomai Userspace`:
    - Enable `Install testsuite`
    - Make sure that `POSIX skin library` is enabled.

Now, build your root filesystem.

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

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

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

Compile a standard Linux kernel

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

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

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

In the kernel configuration interface:

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

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

Compiling with the POSIX RT library

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

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

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

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

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

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

Now compile this program:

```
arm-none-linux-gnueabi-gcc -o rttest rttest.c -lrt
```

Execute the program on the board. Is the clock resolution good or bad? Compare it to the timer tick of your system, as defined by `CONFIG_HZ`.

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

Recompile your kernel, boot your IGEP board with the new version, and check the new resolution. Better, isn't it?

Testing the non-preemptible kernel

Now, do the following tests:

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

Testing the preemptible kernel

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

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

Testing Xenomai scheduling latency

Prepare the kernel for Xenomai compilation:

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

Now, run the kernel configuration interface, and make sure that the below options are enabled, taking your time to read their description:

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

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

```
cd $HOME/felabs/realtime/rttest/buildroot-2013.02/
make host-pkgconf
```

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

```
cd $HOME/felabs/realtime/rttest/nfsroot/root
export PATH=$HOME/felabs/realtime/rttest/buildroot-2013.02/output/host/usr/bin:$PATH
arm-none-linux-gnueabi-gcc -o rttest rttest.c $(pkg-config --libs --cflags libxenomai_posix)
```

Now boot the board with the new kernel.

Run the following commands on the board:

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

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

Run the tests again, compare the results.

Testing Xenomai interrupt latency

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

```
latency -t 2
```


Backing up your lab files

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

End of the training session

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

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

Create a lab archive

Go to the directory containing your felabs directory:

```
cd $HOME
```

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

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

Here is the command:

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

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

