Linux kernel and driver development training

Lab book

Free Electrons
http://free-electrons.com

About this document

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

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

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

Copying this document

Training setup

Download files and directories used in practical labs

Update your system

Time might have elapsed since your system was last updated.

Keep your system up to date:

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

Install lab data

For the different labs in the training, your instructor has prepared a set of data (kernel images, kernel configurations, root filesystems and more). Download the tarball at http://free-electrons.com/labs/embedded_linux.tar.bz2.

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

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

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

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

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

You are now ready to start the real practical labs!

Install extra packages

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

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

More guidelines

Can be useful throughout any of the labs

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

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

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

Kernel source code

Objective: Get familiar with the kernel source code.

After this lab, you will be able to

- Explore the sources in search for files, function headers or other kinds of information...
- Browse the kernel sources with tools like cscope and LXR.

Setup

Go to the `/home/<user>/felabs/linux/modules` directory.

Download and extract the Linux 2.6.39 kernel sources from http://kernel.org.

Please use this exact version, and not 2.6.39.x updates. Doing this will keep it easy to upgrade to 3.0 in the next section.

Apply patches

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

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

Now, download the two Linux patches corresponding to versions 3.0 and 3.0.x.

Apply these patches, check the Makefile file to double check that you have the right version, and rename the source directory to reflect the version change.

We could have downloaded the full archive corresponding to this version. This was just to make you practice with the `patch` command.

Get familiar with the sources

As a Linux kernel user, you will very often need to find which file implements a given function. So, it is useful to be familiar with exploring the kernel sources.

1. Find the Linux logo image in the sources
2. Find who the maintainer of the 3C505 network driver is.
3. Find the declaration of the `platform_device_register()` function.

You may look for all files with `logo` in their name.

Use a kernel source indexing tool

Now that you know how to do things in a manual way, let's use more automated tools.

Try LXR (Linux Cross Reference) at http://lxr.free-electrons.com and choose the Linux version closest to yours.

If you don't have Internet access, you can use `cscope` or `Kscope` instead.

Of course, if your kernel has a significant amount of custom code, or if you are not always connected to the Internet, you can run LXR on your own computer.

As in the previous section, use this tool to find where the `platform_device_register()` is declared, implemented and even used.

Accessing kernel sources with git

Later this week, we will also see another way of accessing kernel sources, with the git source control management tool used by kernel developers.

To save time with the git lab on the last day, let's advance us by cloning the Linus Torvalds' git tree.

First, install required software packages:
```
sudo apt-get install git-core gitk git-email
```

Then, go to the `/home/<user>/felabs/linux/git` directory, and run the below command:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

Replace `git://` by `http://` if your company firewall blocks the git protocol.

Now, just let this command run, from 30 minutes to several hours according to your workstation and network speed.

Kernel – Module development environment

Objective: Setup an NFS based kernel module development environment.

After this lab, you will be able to

- Cross-compile a kernel for the ARM platform
- Boot this kernel on an NFS root filesystem, which is somewhere on your development workstation.

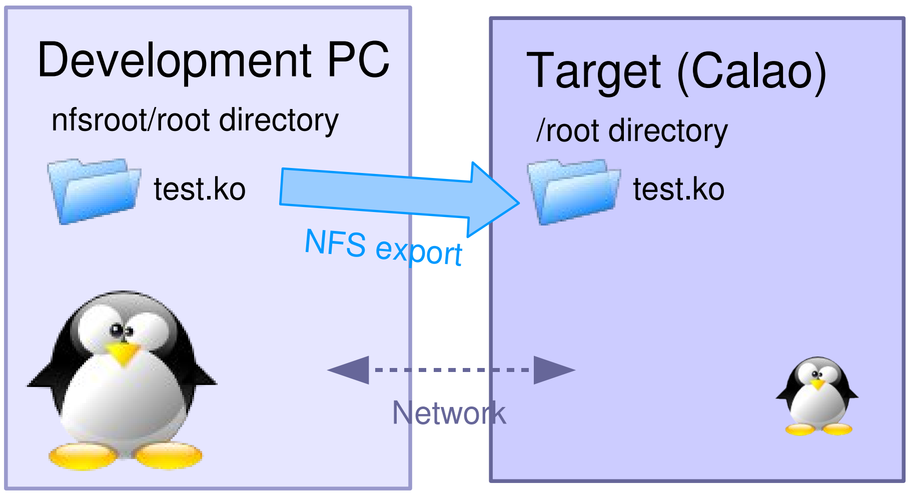
NFS root filesystems are particularly useful to compile modules on your host, and make them directly visible on the target. You longer have to update the root filesystem by hand and transfer it to the target (requiring a shutdown and reboot).

Lab implementation

While developing a kernel module, the developer wants to change the source code, compile and test the new kernel module very frequently. While writing and compiling the kernel module is done the development workstation, the test of the kernel module usually has to be done on the target, since it might interact with hardware specific to the target.

However, flashing the root filesystem on the target for every test is time-consuming and would use the flash chip needlessly.

Fortunately, it is possible to set up networking between the development workstation and the target. Then, workstation files can be accessed through the network by the target, using NFS.



Setup

Stay in the `/home/<user>/felabs/linux/modules` directory.

Install packages needed for this lab:

```
sudo apt-get install libqt4-dev uboot-mkimage
```

`libqt4-dev` is needed for `make xconfig`
`uboot-mkimage` is needed to build the `uImage` file for U-boot.

Cross-compiling toolchain setup

We are going to install a cross-compiling toolchain from Linaro, a very popular source for ARM toolchains (amongst other useful resources for Linux on ARM).

```
sudo add-apt-repository ppa:linaro-maintainers/toolchain
sudo apt-get install gcc-arm-linux-gnueabi
```

Now find out the path and name of the cross-compiler executable by looking at the contents of the package:

```
dpkg -L gcc-arm-linux-gnueabi
```

Note that Linaro toolchains by default generate code for the armv7 instruction set, while our AT91 CPU only supports armv5. This is not a problem, as the kernel Makefile will invoke the cross-compiler with the right instruction set settings.

Kernel configuration

Set the ARCH and CROSS_COMPILE settings for the arm platform and to use your cross-compiler, either by modifying the kernel Makefile or by setting environment variables.

Configure this kernel with the ready-made configuration for boards with the AT91SAM9263 CPU.

If needed, add the configuration options that enable booting on a root filesystem over NFS.

Compile your kernel and generate the uImage kernel image that U-boot needs:

```
make uImage
```

The U-boot bootloader needs the kernel zImage file to be encapsulated in a special container. The kernel Makefile can generate this container for you by running the mkimage tool found in the uboot-mkimage package: make uImage

Setting up the NFS server

Install the NFS server by installing the nfs-kernel-server package. Once installed, edit the /etc/exports file as root to add the following lines, assuming that the IP address of your board will be 192.168.0.100:

```
/home/<user>/felabs/linux/modules/nfsroot 192.168.0.100(rw,no_root_squash,no_subtree_check)
/home/<user>/felabs/linux/character/nfsroot 192.168.0.100(rw,no_root_squash,no_subtree_check)
/home/<user>/felabs/linux/debugging/nfsroot 192.168.0.100(rw,no_root_squash,no_subtree_check)
```

Then, restart the NFS server:

```
sudo /etc/init.d/nfs-kernel-server restart
```

Setting up serial communication with the board

Plug the Calao board on your computer using its USB-A connector. When plugged-in, two serial ports should appear, /dev/ttyUSB0 and /dev/ttyUSB1.

The first serial port of the board, /dev/ttyUSB0, is used for JTAG. The second one, /dev/ttyUSB1, is used to access the DBGU (serial) port of the AT91 CPU.

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

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

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

Run picocom -b 115200 /dev/ttyUSB1, to start a serial communication on /dev/ttyUSB1, with a baudrate of 115200.

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

You should now see the U-Boot prompt:

```
U-Boot>
```

You may need to reset the board (using the tiny reset button close to the USB host connectors).

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

Setting up Ethernet communication

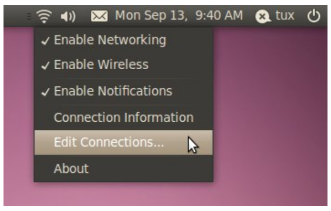
The kernel image will be transferred to the board using the TFTP protocol, which works on top of an Ethernet connection.

To start with, install a TFTP server on your development workstation:

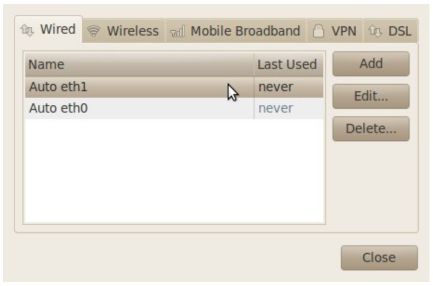
```
sudo apt-get install tftpd-hpa
```

Copy your `uImage` file to the `/var/lib/tftpboot` directory.

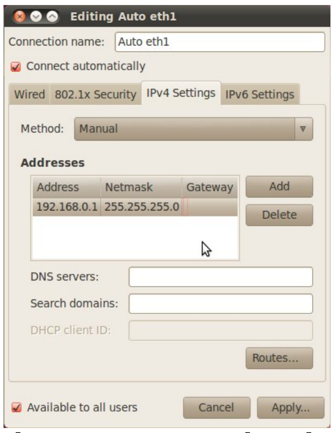
With a network cable, connect the Ethernet port of your board to the one of your computer. If your computer already has a wired connection to the network, your instructor will provide you with a USB Ethernet adapter. A new network interface, probably `eth1` or `eth2`, should appear on your Linux system.



To configure your network interface on the workstation side, click on the Network Manager tasklet on your desktop, and select `Edit Connections`. Select the new wired network connection, and in the `IPv4 Settings` tab, make the interface use a static IP address, like `192.168.0.1` (of course, make sure that this address belongs to a separate network segment from the one of the main company network).



Name	Last Used
Auto eth1	never
Auto eth0	never



Connection name: Auto eth1
✓ Connect automatically
Method: Manual

Address	Netmask	Gateway
192.168.0.1	255.255.255.0	

DNS servers:
Search domains:
DHCP client ID:
✓ Available to all users

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

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

In case the board was previously configured in a different way, we also turn off automatic booting after commands that can be used to copy a kernel to RAM:

```
setenv autostart no
```

To make these settings permanent, save the environment:

```
saveenv
```

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

```
tftp 0x21000000 textfile.txt
```

This should download the file `textfile.txt` from your development workstation into the board's memory at location `0x21000000` (this location is part of the board DRAM). You can verify that the download was successful by dumping the contents of the memory:

```
md 0x21000000
```

Boot the system

First, boot the board to the U-Boot prompt.

Before booting the kernel, we need to tell it that the root filesystem should be mounted over NFS, by setting some kernel parameters.

Use the following U-Boot command to do so (in just 1 line):

```
setenv bootargs root=/dev/nfs ip=192.168.0.100 nfsroot=192.168.0.1:
/home/<user>/felabs/linux/modules/nfsroot
```

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

Now, download the kernel image through tftp:

```
tftp 0x21000000 uImage
```

Now, boot your kernel:

```
bootm 0x21000000
```

If everything goes right, you should reach a shell prompt. Otherwise, check your setup or ask your instructor for details.

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`.

Kernel – Writing modules

Objective: create a simple kernel module

After this lab, you will be able to

- Compile and test standalone kernel modules, which code is outside of the main Linux sources.
- Write a kernel module with several capabilities, including module parameters.
- Access kernel internals from your module.
- Setup the environment to compile it
- Create a kernel patch

Setup

Stay inside the `/home/<user>/felabs/linux/modules` directory.

Boot your board again, as you did in the previous lab.

Writing a module

Go to the `nfsroot/root` directory. All the files you generate there will also be visible from the target. That's great to load modules!

Create a `hello_version.c` file implementing a module which displays this kind of message when loaded:

You may just start with a module that displays a hello message, and add version information later.

```
Hello Master. You are currently using Linux <version>.
```

... and displays a goodbye message when unloaded.

Caution: you must use a kernel variable or function to get version information, and not just the value of a C macro. Otherwise, you will only get the version of the kernel you used to build the module.

Suggestion: you can look for files in kernel sources which contain `version` in their name, and see what they do.

Building your module

The current directory contains a `Makefile` file, which lets you build modules outside a kernel source tree.

Actually, you don't need complete kernel sources to build a module. A build directory is enough.

Compile your module.

Testing your module

Load your new module file. Check that it works as expected. Until this, unload it, modify its code, compile and load it again as many times as needed.

Run a command to check that your module is on the list of loaded modules. Now, try to get the list of loaded modules with only the `cat` command.

Adding a parameter to your module

Add a `who` parameter to your module. Your module will say “Hello <who>” instead of “Hello Master”.

Compile and test your module by checking that it takes the `who` parameter into account when you load it.

Adding time information

Improve your module, so that when you unload it, it tells you how many seconds elapsed since you loaded it.

You can use the `do_gettimeofday()` function to achieve this.

You may search for other drivers in the kernel sources using the `do_gettimeofday()` function. Looking for other examples always helps!

Following Linux coding standards

Your code should adhere to strict coding standards, if you want to have it one day merged in the mainline sources. One of the main reasons is code readability. If anyone used one's own style, given the number of contributors, reading kernel code would be very unpleasant.

Fortunately, the Linux kernel community provides you with a utility to find coding standards violations.

Run the `scripts/checkpatch -h` command in the kernel sources, to find which options are available.

Now, run:

```
scripts/checkpatch --file --no-tree <path>/hello_version.c
```

See how many violations are reported on your code. If there are indenting errors, you can first run your code through the `ident` command:

```
sudo apt-get install indent
indent -linux hello_version.c
```

You can now compare the indented file with the original:

```
sudo apt-get install meld
meld hello_version.c~ hello_version.c
```

Now, get back to `checkpatch.pl` and fix your code until there are no errors left.

Adding the hello_version sources to the kernel sources

Add your module sources to the `drivers/misc/` directory in your kernel sources. Of course, also modify kernel configuration and building files accordingly, so that you can select your module in `make xconfig` and have it compiled by the `make` command.

Configure your kernel with the config file corresponding to your running kernel. Now check that the configuration interface shows your new driver and lets you configure it as a module.

Run the `make` command and make sure that the code of your new driver is getting compiled. Then, install your kernel module using `make modules_install`. Beware, the modules should be installed in the root filesystem of the target, not in the root filesystem of your development workstation!

You may also wait for compiling to be over and look for a new `hello_version.ko` file, but this may take much more time.

Create a kernel patch

You can be proud of your new module! To be able to share it with others, create a patch which adds your new files to the mainstream kernel.

Test that your patch file is compatible with the `patch` command by applying it to unmodified kernel sources.

Kernel - Accessing I/O memory and ports

Objective: read / write data from / to a hardware device

Throughout the upcoming labs, we will implement a character driver allowing to write data to the serial port of the CALAO board, and to read data from it.

After this lab, you will be able to

- Use ssh to get a remote terminal to an embedded Linux system
- Access a console through the network, typically when a serial or graphical console cannot be used.
- Practice with I/O ports and I/O memory to control the device and exchange data with it.

Setup

Go to the `/home/<user>/felabs/linux/character` directory.

As in the *Module development environment* lab, we will use a the CALAO board booted from NFS.

As in the previous labs, the target IP address will be `192.168.0.100`, and the host address will be `192.168.0.1`.

Extract the latest Linux 3.0.x kernel sources in the current directory, and configure them with the default configuration for at91sam9263 boards.

In this lab, we will develop our own driver for the board's serial port. The consequence is that we will have to disable the standard AT91 serial port driver and will thus loose the serial console.

Instead of running commands through a shell on the serial line, we will access our target through SSH, a secure shell over the network.

To replace the serial console and still get kernel messages (in particular kernel fault messages) we will use a *netconsole*, a mechanism that allows to get console messages over the network.

So, configure your kernel with:

- Root over NFS support
- Loadable module support
- Netconsole support (`CONFIG_NETCONSOLE`)

You will also have to update the kernel command line so that Linux loads the root filesystem over NFS from `/home/<user>/felabs/linux/character/nfsroot`.

The Dropbear SSH server is already installed on your target, and is started automatically.

Boot tests

We are first going to make sure that your new kernel boots fine and that the SSH connection works well, before disabling the serial driver and the serial console. NFS configuration issues are frequent, and they would be more difficult to fix if we have no console left.

So, boot your new kernel and try to connect to your board with `ssh`:

```
ssh root@192.168.0.100
```

The root password is empty, just press Enter. Good job!

Disabling the serial driver and console

Now that everything works, rebuild your kernel without the serial port driver (in `Device Drivers → Character Drivers → Serial drivers → AT91 / AT32 on-chip serial port support`). Update your kernel.

You also need to add the following option to the kernel command line, to enable the network console:

```
netconsole=4444@192.168.0.100/eth0,5555@192.168.0.1/
```

To get the console output, we will use the `netcat` command on the host to listen for UDP packets on port 5555:

```
netcat -u -l 192.168.0.1 5555
```

Now, boot your new kernel. You should see kernel messages only in the `netcat` output (after a little while).

Driver source code

Now go to the `nfsroot/root` directory, which will contain the kernel module source code.

You have a ready-made `Makefile` to compile your module, and a template `serial.c` module which you will fill with your own code. Modify the `Makefile` so that it points to your kernel sources.

Device initialization

In the module initialization function, start by reserving the I/O memory region starting at address `(AT91_BASE_SYS + AT91_DBGU)`, for a size of `SZ_512` (512 bytes). The `AT91_*` constants are already defined in Linux kernel headers.

Compile your module, load it and make sure that this memory region appears in `/proc/iomem`.

Now, obtain a virtual address corresponding to the start of this memory area.

Don't forget to undo all the above in the module exit function!

Standalone write routine

Implement a C routine taking one character as a parameter and writing it to the serial port, using the following steps:

1. Wait until the `ATMEL_US_TXRDY` bit gets set in the `ATMEL_US_CSR` register (`ATMEL_US_CSR` is an offset in the I/O memory region previously remapped). You can busy-wait for this condition to happen. In the busy-wait loop, you can call the `cpu_relax()` kernel function to relax the CPU during the wait.
2. Write the character to the `ATMEL_US_THR` register.

All the registers we will read and write are 32 bit wide.

See `Documentation/volatile-considered-harmful.txt` in kernel documentation and http://j.mp/q6BcGv if you are curious about the use of `cpu_relax()`.

Add a call to this routine from your module init function. Recompile your module and load it on the target. You should see the corresponding character in Minicom, still showing what was written to the serial line by the board.

Driver sanity check

Remove your module and try to load it again. If the second attempt to load the module fails, it is probably because your driver doesn't properly free the resources it allocated or register, either at module exit time, or after a failing during the module init function.

Check and fix your module init and exit functions if you have such a problem.

Kernel - Output-only character driver

Objective: implement the write part of a character driver

After this lab, you will be able to

- Write a simple character driver, allowing to write data to the serial port of your CALAO board.
- Write simple `file_operations` functions for a device, including `ioctl` controls.
- Copy data from user memory space to kernel memory space and eventually to the device.
- You will practice kernel standard error codes a little bit too.

Setup

You must have completed the previous lab before.

Stay in the `/home/<user>/felabs/linux/character` directory.

Major number registration

Find an available character device major number on your virtual system.

Modify the `serial.c` file to register this major number in your new driver. Compile your module, load it, and check that it effectively registered the major number you chose.

Simplistic character driver

Add code to register a character driver with your major number, and the empty file operation functions which already exist in `serial.c`. Also create the corresponding `/dev/serial` device file.

Now, add code to your `write` function, to copy user data to the serial port, writing characters one by one.

Once done, compile and load your module. Test that your `write` function works properly by using:

```
      echo “test” > /dev/serial
```

The `“test”` string should appear on the remote side (i.e in the Minicom process connected to `/dev/ttyUSB1`).

You'll quickly discover than newlines do not work properly. To fix this, when the userspace application sends `“\n”`, you must send `“\n\r”` to the serial port.

Driver sanity check

Once again, make sure that you can load your module again after removing it.

Ioctl operation

We would like to maintain a counter of the number of characters written through the serial port. So we need to implement two `unlocked_ioctl()` operations:

- `SERIAL_RESET_COUNTER`, which as its name says, will reset the counter to zero.
- `SERIAL_GET_COUNTER`, which will return in a variable passed by address the current value of the counter;

Two already-compiled test applications are already available in the `nfsroot/root/` directory, with their source code. They assume that `SERIAL_RESET_COUNTER` is ioctl operation 0 and that `SERIAL_GET_COUNTER` is ioctl operation 1.

Kernel - Sleeping and handling interrupts

Objective: learn how to register and implement a simple interrupt handler, and how to put a process to sleep and wake it up at a later point

During this lab, you will

- Register an interrupt handler for the serial controller of the Calao board
- See how Linux handles shared interrupt lines
- Implement the `read()` operation of the serial port driver to put the process to sleep when no data is available
- Implement the interrupt handler to wake-up the sleeping process waiting for received characters
- Handle communication between the interrupt handler and the `read()` operation.

Setup

This lab is a continuation of the *Output-only character driver* lab, so we'll re-use the code in `/home/<user>/felabs/linux/character`. Your Calao board should boot over NFS and mount `/home/<user>/felabs/linux/character/nfsroot/` as the root filesystem.

Register the handler

First, declare an interrupt handler function. Then, in the module initialization function, register this handler to IRQ number `AT91_ID_SYS`. Note that this IRQ is shared, so the appropriate flags must be passed at registration time.

Then, in the interrupt handler, just print a message and return `IRQ_NONE` (to tell the kernel that we haven't handled the interrupt).

Compile and load your module. Look at the kernel logs: they are full of our message indicating that interrupts are occurring, even if we are not receiving from the serial port! It shows you that interrupt handlers on shared IRQ lines are all called every time an interrupt occurs.

Enable and filter the interrupts

In fact, at the moment, reception and interrupts are not enabled at the level of the serial port controller. So in the initialization function of the module:

- Write `ATMEL_US_RSTSTA | ATMEL_US_RSTRX` to the `ATMEL_US_CR` register;
- Write `ATMEL_US_TXEN | ATMEL_US_RXEN` to the `ATMEL_US_CR` register;
- Write `ATMEL_US_RXRDY` to the `ATMEL_US_IER` register (IER stands for Interrupt Enable Register).

Now, in our interrupt handler we want to filter out the interrupts that come from the serial controller. To do so, read the value of the

ATMEL_US_CSR register and the value of the ATMEL_US_IMR register. If the result of a *binary and* operation between these two values is different from zero, then it means that the interrupt is coming from our serial controller.

If the interrupt comes from our serial port controller, print a message and return IRQ_HANDLED. If the interrupt doesn't come from our serial port controller, just return IRQ_NONE without printing a message.

Compile and load your driver. Have a look at the kernel messages. You should no longer be flooded with interrupt messages.

Start minicom on /dev/ttyUSB1. Press one character (nothing will appear since the target system is not echoing back what we're typing). Then, in the kernel log, you should see the message of our interrupt handler. If not, check your code once again and ask your instructor for clarification!

Read the received characters

You can read the received characters by reading the ATMEL_US_RHR register using readl(). It must be done in code that loops until the ATMEL_US_RXRDY bit of the ATMEL_US_CSR register goes back to zero. This method of operation allows to read several characters in a single interrupt.

Note that our hardware doesn't give us any special register to acknowledge interrupts. What happens is that interrupts are acknowledged (allowing more interrupts to be sent in the future), when the driver accesses the ATMEL_US_RHR register to read each character.

For each received character, print a message containing the character.

Compile and load your driver. From minicom on /dev/ttyUSB1 on the host, send characters to the target. The kernel messages on the target should properly tell you which characters are being received.

Sleeping, waking up and communication

Now, we would like to implement the read() operation of our driver so that a userspace application reading from our device can receive the characters from the serial port.

First, we need a communication mechanism between the interrupt handler and the read() operation. We will implement a very simple circular buffer. So let's declare a global buffer in our driver:

```
#define SERIAL_BUFSIZE 16
static char serial_buf[SERIAL_BUFSIZE];
```

Two integers that will contain the next location in the circular buffer that we can write to, and the next location we can read from:

```
static int serial_buf_rd, serial_buf_wr;
```

In the interrupt handler, store the received character at location serial_buf_wr in the circular buffer, and increment the value of serial_buf_wr. If this value reaches SERIAL_BUFSIZE, reset it to zero.

In the read() operation, if the serial_buf_rd value is different from the serial_buf_wr value, it means that one character can be read from the circular buffer. So, read this character, store it in the userspace buffer, update the serial_buf_rd variable, and return to userspace (we will only read one character at a time, even if the userspace application requested more than one).

Now, what happens in our `read()` function if no character is available for reading (i.e, if `serial_buf_wr` is equal to `serial_buf_rd`)? We should put the process to sleep!

To do so, declare a global wait queue in our driver, named for example `serial_wait`. In the `read()` function, use `wait_event_interruptible()` to wait until `serial_buf_wr` is different from `serial_buf_rd`. And in the interrupt handler, after storing the received characters in the circular buffer, use `wake_up()` to wake up all processes waiting on the wait queue.

Compile and load your driver. Run `cat /dev/serial` on the target, and then in Picocom on the development workstation side, type some characters. They should appear on the remote side if everything works correctly!

Don't be surprised if the keys you type in Picocom don't appear on the screen. This happens because they are not echoed back by the target.

You could modify your driver so that the read function always writes received characters back to the serial line. This way, you would see what you type.

Kernel - Locking

Objective: practice with basic locking primitives

During this lab, you will

- Practice with locking primitives to implement exclusive access to the device.

Setup

Stay in the `/home/<user>/felabs/linux/character` directory.

You need to have completed the previous two labs to perform this one.

Boot your board with the same NFS environment as before, and load your serial module.

Adding appropriate locking

We have two shared resources in our driver:

- The buffer that allows to transfer the read data from the interrupt handler to the read() operation.
- The device itself. It might not be a good idea to mess with the device registers at the same time and in two different contexts.

Therefore, your job is to add a spinlock to the driver, and use it in the appropriate locations to prevent concurrent accesses to the shared buffer and to the device.

Please note that you don't have to prevent two processes from writing at the same time: this can happen and is a valid behavior. However, if two processes write data at the same time to the serial port, the serial controller should not get confused.

Kernel – Kernel debugging mechanisms and kernel crash analysis

Objective: Use kernel debugging mechanisms and analyze a kernel crash.

pr_debug() and dynamic debugging

Add a `pr_debug()` call in the `write()` operation that shows each character being written (or its hexadecimal representation) and add a similar `pr_debug()` call in your interrupt handler to show each character being received.

Check what happens with your module. Do you see the debugging messages that you added ? Your kernel probably has neither `CONFIG_DEBUG` nor `CONFIG_DYNAMIC_DEBUG` set, so you shouldn't see any message.

Now, recompile your kernel with `CONFIG_DYNAMIC_DEBUG` and reboot. The dynamic debug feature can be configured using *debugfs*, so you'll have to mount the *debugfs* filesystem first. Then, after reading the dynamic debug documentation in the kernel sources, do the following things :

- List all available debug messages in the kernel
- Enable all debugging messages of your serial module, and check that you indeed see those messages.
- Enable just one single debug message in your serial module, and check that you see just this message and not the other debug messages of your module.

Now, you have a good mechanism to keep many debug messages in your drivers and be able to selectively enable only some of them.

debugfs

Since you have enabled *debugfs* to control the dynamic debug feature, we will also use it to add a new *debugfs* entry. Modify your driver to add :

- A directory called “serial” in the *debugfs* filesystem
- And file called “counter” inside the “serial” directory of the *debugfs* filesystem. This file should allow to see the contents of the “counter” variable of your module.

Recompile and reload your driver, and check that in `/sys/kernel/debug/serial/counter` you can see the amount of characters that have been transmitted by your driver.

Kernel crash analysis

Setup

Go to the `/home/<user>/felabs/linux/debugging` directory.

Make sure that your kernel has the following options enabled:

- Support for root filesystem over NFS support

- The `CONFIG_DEBUG_INFO` configuration option, (`Kernel Hacking` section) which makes it possible to see source code in the disassembled kernel
- Disable the `CONFIG_ARM_UNWIND` configuration option (`Kernel Hacking` section). This option enables a new mechanism to handle stack backtraces, but this new mechanism is not yet as functional and reliable as the old mechanism based on frame pointers. In our case, with our board, you get a backtrace only if this option is disabled.

The `nfsroot/` directory is the root filesystem.

Compile the `drvbroken` module provided in `nfsroot/root/drvbroken`, after modifying the Makefile so that `KDIR` properly points to your kernel source tree.

Run the target system on the Calao board, and load the `drvbroken` kernel module. See it crashing in a nice way.

Analyzing the crash message

Analyze the crash message carefully. Knowing that on ARM, the `pc` register contains the location of the instruction being executed, find in which function does the crash happens, and what the function call stack is.

Using LXR (for example http://lxr.free-electrons.com) or the kernel source code, have a look at the definition of this function. This, with a careful review of the driver source code should probably be enough to help you understand and fix the issue.

Further analysis of the problem

If the function source code is not enough, then you can look at the disassembled version of the function, either using:

```
arm-linux-gnueabi-objdump -S linux-3.0.x/vmlinux >
vmlinux.disasm
```

or:

```
sudo apt-get install gdb-multiarch
```

```
gdb-multiarch linux-3.0.x/vmlinux
(gdb) set arch arm
(gdb) set gnutarget elf32-littlearm
(gdb) disassemble function_name
```

`gdb-multiarch` is a new package supporting multiple architectures at once.

If you have a cross toolchain including `gdb`, you can also run `arm-linux-gdb` directly.

Then find at which exact instruction the crash occurs. The offset is provided by the crash output, as well as a dump of the code around the crashing instruction.

Of course, analyzing the disassembled version of the function requires some assembly skills on the architecture you are working on.

Kernel – Serial controller device driver programming

Objective: Develop a serial device driver for the AT91SAM9263 CPU from scratch.

Warning

In this lab, we are going to re-implement a driver that already exists in the Linux kernel tree. Since the driver already exists, you could just copy the code, compile it, and get it to work in a few minutes. However, the purpose of this lab is to re-create this driver from scratch, taking the time to understand all the code and all the steps. So please play the game, and follow our adventure of creating a serial driver from scratch!

Setup

Go to the `/home/<user>/felabs/linux/character` directory. It contains the root filesystem that you will mount over NFS to work on this lab. Re-use the setup instructions of the lab on Character Device Drivers to get a kernel without the serial port driver and with Network Console support.

Basic module

The serial core cannot be compiled inside the kernel without an in-tree kernel driver. Therefore, for this lab, we will work directly inside the kernel source tree and not using an external module.

To do so:

- Create a basic module in `drivers/serial/fedrv.c` with just the init and cleanup functions ;
- Add a new configuration option in `drivers/serial/Kconfig`. Don't forget to select `SERIAL_CORE` in this option ;
- Update `drivers/serial/Makefile` to make sure your driver gets compiled when your new option is selected

Compile your new driver as a module, and after the kernel compilation, run:

```
make INSTALL_MOD_PATH=/path/to/nfsroot modules_install
```

To install the modules (`serial_core` and your driver) into the root filesystem. Then try to load/unload your module on the target using `modprobe`. If you're successful, we can now start working on the driver itself.

Register the UART driver

Instantiate an `uart_driver` structure with the following values:

- `owner`, `THIS_MODULE`
- `driver_name`, `“fedrv”` or any other string
- `dev_name`, `“ttyS”`
- `major`, `TTY_MAJOR` (this is the usual major for TTY devices)

- minor, 64 (this is the usual minor for TTY serial devices, see Documentation/devices.txt in the kernel source tree)
- nr, 1 (we will support only one port)

In the init function, register the UART driver with uart_register_driver() and in the cleanup function, unregister it with uart_unregister_driver().

Integration in the device model

To get notifications of the UART devices that exist on our board, we will integrate our driver in the device model.

To do, so, first instantiate a platform_driver structure, with pointers to the probe() and remove() methods (they can be left empty at the moment). The driver name must be "atmel_usart" to match the device definitions in arch/arm/mach-at91/.

You should mark the probe function with __devinit and the remove function with __devexit. The remove operation should be declared as follows:

```
        .remove = __devexit_p(fedrv_remove)
```

So that if the driver is statically compiled, the fedrv_remove() function is not compiled in and the .remove pointer is NULL.

Then, in the init and cleanup functions of the module, register and unregister the platform driver using platform_driver_register() and platform_driver_unregister().

Finally, you need to make a small modification to the kernel. Currently, the "atmel_usart" platform devices are only added if the Atmel serial port driver is compiled in. However, since we disabled this driver (because we are re-implementing it), we must modify a little the board code. So, in arch/arm/mach-at91/at91sam9263_devices.c, replace:

```
#if defined(CONFIG_SERIAL_ATMEL)
```

by

```
#if 1
```

Then, recompile your kernel, re-flash it, and test your new module. You should see your probe() function being called (after adding a simple printk() in it). And in /sys/devices/platform/, you should see the device atmel_usart.0. This directory contains a symbolic link driver to the atmel_usart driver. If you follow this symbolic link, you should discover that the atmel_usart driver is implemented by the fedrv module. Congratulations!

Registering the port

Now, it's time to implement the probe() and remove() functions. Before that, we need a few definitions:

- Declare a global uart_port structure, that will be used to contain the informations about the single port we will manage;
- Declare an empty uart_ops structure.

Then, in the `probe()` operation:

- Make sure `pdev->id` is `0` (we only want to handle the first serial port). If it's not zero, bail out with `-ENODEV`
- Initialize the fields of the `uart_port` structures
    - `->ops` should point to the `uart_ops` structure
    - `->dev` should point to the `struct device` embedded in the platform device structure. So `&pdev->dev` should work
    - `->line` should be the serial port number, i.e `0` or `pdev->id`
- Register the port with `uart_add_one_port()`
- Associate the port pointer to the platform device structure using `platform_set_drvdata()`. This will make it easy to find the port structure from the platform device structure in the `remove()` operation.

In the `remove()` method:

- Get the port structure from the platform device structure using `platform_get_drvdata()`
- Unregister the port with `uart_remove_one_port()`.

Now, when testing your driver, in `/sys/devices/platform/atmel_usart.0/`, you should have a `tty` directory, which itself contains a `ttyS0` directory. Similarly, if you go in `/sys/class/tty/ttyS0`, you should see that the `ttyS0` device is handled by `atmel_usart.0`. Good!

Polled mode transmission

To keep our driver simple, we will implement a very simple polled-mode transmission model.

In the `probe()` operation, let's define a few more things in the port structure:

- `->fifosize`, to 1 (this is hardware-dependent)
- `->iotype` should be `UPIO_MEM` because we are accessing the hardware through memory-mapped registers
- `->flags` should be `UPF_BOOT_AUTOCONF` so that the `config_port()` operation gets called to do the configuration
- `->mapbase` should be `pdev->resource[0].start`, this is the address of the memory-mapped registers
- `->membase` should be set to `data->regs`, where data is the device-specific platform data associated to the device. In our case, it's a `atmel_uart_data` structure, available through `pdev->dev.platform_data`. In the case of the first serial port `data->regs` is non-zero and contains the virtual address at which the registers have been remapped. For the other serial ports, we would have to `ioremap()` them.

Then, we need to create stubs for a fairly large number of operations. Even if we don't implement anything inside these operations for the moment, the `serial_core` layer requires these operations to exist:

- `tx_empty()`
- `start_tx()`
- `stop_tx()`
- `stop_rx()`
- `type()`
- `startup()`
- `shutdown()`
- `set_mctrl()`
- `set_termios()`
- `release_port()`
- `request_port()`
- `config_port()`

First, let's implement what's related to setting and getting the serial port type:

- In the `config_port()` operation, if `flags & UART_CONFIG_TYPE` is true, then set `port->type = PORT_ATMEL`. There is a global list of serial port types, and we are re-using the existing definition.
- In the `type()` operation, if `port->type` is `PORT_ATMEL` return a string like “`ATMEL_SERIAL`”, otherwise return `NULL`.

Now, for the transmission itself, we will first implement `tx_empty()`. In this function, read the register `ATMEL_US_CSR` from the hardware (note: the virtual base address of the registers is in `port->membase`). If bit `ATMEL_US_TXEMPTY` is set, it means that the port is ready to transmit, therefore return `TIOCSER_TEMT`, otherwise return 0.

Then, the `start_tx()` function will do the transmission itself. Iterate until the transmission buffer is empty (use `uart_circ_empty()`) and do:

- call an auxiliary function that prints one character
- update the tail pointer of the transmission buffer
- increment `port->icount.tx`

The auxiliary function should wait until bit `ATMEL_US_TXRDY` gets set in the `ATMEL_US_CSR` register, and then send the character through the `ATMEL_US_THR` register.

Then, compile and load your driver. You should now be able to do `echo “foo” > /dev/ttyS0`.

Implementing reception

The last part to make our driver usable is to implement reception.

We first need to modify the `probe()` method to set `port->irq` to `pdev->resource[1].start` so that we fetch the IRQ number from the board-specific platform device definition.

Then, in the `startup()` operation, do the following steps:

- Disable all interrupts in the serial controller by writing `~0UL` to the `ATMEL_US_IDR` register
- Register the IRQ channel `port->irq` to an interrupt handler `fedrv_interrupt()`. Pass `port` as the `dev_id` so that we get a pointer to the port in the interrupt handler. Make it a shared interrupt.
- Reset the serial controller by writing `ATMEL_US_RSTSTA | ATMEL_US_RSTRX` to the `ATMEL_US_CR` register
- Enable transmission and reception by writing `ATMEL_US_TXEN | ATMEL_US_RXEN` to the `ATMEL_US_CR` register
- Enable interrupts on reception by writing `ATMEL_US_RXRDY` to the `ATMEL_US_IER` register

Similarly, in the `shutdown()` operation, do:

- Disable all interrupts by writing `~0UL` to the `ATMEL_US_IDR` register
- Free the IRQ channel using `free_irq()`.

Then, in the interrupt handler, do the following:

- Read the `ATMEL_US_CSR` register to get the controller status and perform the logical and of this value with the enabled interrupts by reading the `ATMEL_US_IMR` register. If the resulting value is 0, then the interrupt was not for us, return `IRQ_NONE`.
- If the result value has bit `ATMEL_US_RXRDY` set, call an auxiliary function `fedrv_rx_chars()` to receive the characters.

Finally, we have to implement the `fedrv_rx_chars()` function. This function should read the `ATMEL_US_CSR` register, and while `ATMEL_US_RXRDY` is set in this register, loop to read characters the following way:

- Read one character from the `ATMEL_US_RHR` register
- Increment `port->icount.rx`
- Call `uart_insert_char()` with the value of the status register, overrun to `ATMEL_US_OVRE`, and the flag set to `TTY_NORMAL` (we don't handle break characters, frame or parity errors, etc. for the moment)

Once all characters have been received, we must tell the upper layer to push these characters, using `tty_flip_buffer_push()`.

Now, if you do cat `/dev/ttyS0`, you should be able to receive characters from the serial port. By default, a `ttyS` is opened in the so-called “canonical” mode, so the characters are sent to the reading process only after entering a newline character.

You can also try to run the program that will display the login prompt and then a shell:

```
/sbin/getty -L ttyS0 115200 vt100
```

Go back to your Minicom, you should be able to login normally, but using your own serial driver!

Improvements

Of course, our serial driver needs several improvements:

- Real implementation of `set_termios()` and `set_mctrl()`
- Usage of interrupts for transmission
- Console support for early messages
- Support of several serial ports
- Handle parity and frame errors properly
- Support break characters and SysRq
- Use of DMA for transmission and reception
- etc.

Power management

Objective: practice with standard power management interfaces offered by Linux

After this lab, you will be able to

- Suspend and resume your Linux system
- Change the CPU frequency of your system

Setup

Lab data are available in `/home/<user>/felabs/powermgt/usage/`.

Download and extract the latest update to the Linux 2.6.31 kernel.

Suspend/resume support for the Calao board is already included in this kernel.

Cpu frequency scaling support for this hardware was developed by Free Electrons and is not yet part of the mainline kernel. Therefore, before compiling a recent 2.6.31 kernel, you'll have to apply the three patches in the `data/` directory of this lab:

- The first patch implements the CPU frequency driver itself, which allows to change the frequency on the AT91SAM9263 CPU
- The second patch adds CPU frequency support to the serial port driver. When the CPU clock is changed, the divisors for the baud rate generator must be modified. This is what this patch does.
- The third patch adds CPU frequency support to the Ethernet controller driver for similar reasons.

Configure your kernel with CPU Frequency scaling support, with the CPU Frequency driver for AT91, and for the different cpufreq governors.

Then, compile this kernel, and boot the system over NFS to the root filesystem included in
`/home/<user>/felabs/powermgt/usage/nfsroot/`.

Suspend and resume

To suspend to RAM the Calao board, run :

```
echo mem > /sys/power/state
```

The Calao board will then put itself in a low power-consumption mode, as the inactivity of most LEDs will show.

To resume the Calao board, push the User button. After a short time, the board will be usable again.

CPU frequency control

Linux has a `cpufreq` driver to control CPU frequency. Of course, it can only switch between the limited number of operating states that your CPU and board can support.

This interface can be controlled by userspace. This means it allows you to let the system user tune it from a graphical front-end, for example.

Go to the `/sys/devices/system/cpu/cpu0/cpufreq/` directory, and see what available files are.

Check what the current `cpufreq` governor is.

Find what the allowed frequencies are on your system.

Now look at the files which offer write permission. These are the ones you can use to control `cpufreq`.

Switch to the `userspace` governor, the one that disables the kernel autopilot. Now, set the frequency of `cpu0` to the maximum one. View the `scaling_cur_freq` file to check that the frequency is the one you expected.

With the `userspace` governor enabled, you can implement your own, custom CPU frequency control based on your own criteria. You could check the system temperature, for example, and if it gets hotter than a specified threshold, you could slow down the frequency. You could also let a time critical process bump the frequency to the maximum value. You can see that in userspace, **you** are the governor.

Change the governor to `performance`, check the current frequency, and change to `powersave` and check the frequency again.

You can also select the `ondemand` governor, add some load to your target by running `ping -f target-ip` (ping in flood mode) from your PC and see the cpu frequency increase when your system gets pinged.

Using PowerTop

On your development PC. Install the nice PowerTop tool contributed by Intel:

```
sudo apt-get install powertop
```

Run the `powertop` command, and see it display statistics, and list the top processes that cause you CPU to wake up from a deeper sleep state, causing it to consume more power. You could use this interface to find power management bugs in the applications running on your system.

If you're using a laptop, remove the AC power for a while. This gives you access to live power estimates from ACPI.

Also follow the tips that PowerTop gives you to conserve power, and try to make your system consume as little power as possible.

Compare your power estimates with other people in the classroom, and try to achieve the best results. Any technique can be used!

Thanks to Linaro, PowerTop is also available on ARM now. See https://wiki.linaro.org/WorkingGroups/PowerManagement/Doc/Powertop .

So, if your embedded architecture has CPUidle support, you could try this utility on it.

If your embedded architecture has CPUidle support, even if you didn't compile `powertop`, you can still access idle state statistics by looking at the files in `/sys/devices/system/cpu/cpu<n>/cpuidle`

Lab – Git

Objective: use the basic Git features

After this lab, you will be able to

- Clone a Git repository
- Explore the history of a Git repository
- Make changes in your own branch
- Generate the patches corresponding to your own branch

Setup

Go to `/home/<user>/felabs/linux/git/`

This lab assumes that you already installed git software and cloned the Linus Torvalds' git tree. See our "Kernel source code" lab for details (http://free-electrons.com/doc/training/linux-kernel/).

Configuring Git

Configure your name and email address in git with `git config`.

Clone a repository

We already cloned Linus Torvalds' git tree, but it is useful to know how to do it again. Go to http://git.kernel.org and make sure you know how to find the `git://` URL of his Linux 2.6 tree.

That's a very easy way of finding this URL again. You can copy this URL from your browser.

Cloning downloaded quite a lot of data, but then at the end, we have the full history of the Linux kernel (since the kernel developers started to use Git, around kernel 2.6.12). We can access and explore this history offline.

Exploring the history

With `git log`, look at the list of changes that have been made on the scheduler.

With `git log`, look at the list of changes and their associated patches, that have been made on the ATMEL serial driver (`drivers/serial/atmel_serial.c`) between the versions 2.6.29 and 2.6.30 of the kernel.

With `git diff`, look at the differences between `fs/jffs2/` (which contains the JFFS2 filesystem driver) in 2.6.29 and 2.6.30.

With `gitk`, look at the full history of the UBIFS filesystem (in `fs/ubifs/`).

On the *gitweb* interface of Linus Torvalds tree, available at http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git, search all commits that have been done by Free Electrons (hint: use the search engine by author).

Make your changes

Create your own branch with `git branch` and then move to it with `git checkout`.

Make a dummy change to the `MAINTAINERS` file, and commit your change. Look at the difference between the master branch and your branch (with `git log`, `git diff` and `gitk`).

Then, edit `init/main.c`. In the function `start_kernel()`, after the call to `printk()` to print the `linux_banner` variable, add a call to `printk()` to print your own message. Commit your change.

Share your changes

Generate the patch series corresponding to your two changes using `git format-patch`.

Configure your SMTP server using:

```
git config --global sendemail.smtpserver smtp.company.com
```

And then send the patches to yourself using `git send-email`.

Tracking another tree

Say you want to work on the realtime Linux tree, so we'll add this tree to the trees you're tracking:

```
git remote add realtime
git://git.kernel.org/pub/scm/linux/kernel/git/rostedt/lin
ux-2.6-rt.git
```

A `git fetch` will fetch the data for this tree. Of course, Git will optimize the storage, and will no store everything that's common between the two trees. This is the big advantage of having a single local repository to track multiple remote trees, instead of having multiple local repositories.

We can then switch to the master branch of the realtime tree:

```
      git checkout realtime/master
```

Or look at the difference between the scheduler code in the official tree and in the realtime tree:

```
      git diff master..realtime/master kernel/sched.c
```

Backing up your lab files

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

End of the training session

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

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

Create a lab archive

Go to the directory containing your `felabs` directory:
```
cd $HOME
```

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

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

Here is the command:

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

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