Linux kernel and driver development training

Lab Book

Free Electrons
http://free-electrons.com

November 29, 2013

About this document

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

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

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

Copying this document

Corrections, suggestions, contributions and translations are welcome!

Training setup

Download files and directories used in practical labs

Install lab data

For the different labs in the training, your instructor has prepared a set of data (kernel images, kernel configurations, root filesystems and more). Download and extract its tarball from a terminal:

```
cd
wget http://free-electrons.com/doc/training/linux-kernel/labs.tar.xz
sudo tar Jvxf labs.tar.xz
sudo chown -R <user>.<user> felabs
```

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

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

You are now ready to start the real practical labs!

Install extra packages

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

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

More guidelines

Can be useful throughout any of the labs

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

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

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

Downloading kernel source code

Get your own copy of the mainline Linux kernel source tree

Setup

Go to the `$HOME/felabs/linux/src` directory.

Installing git packages

First, let's install software packages that we will need throughout the practical labs:

```
sudo apt-get install git gitk git-email
```

Git configuration

After installing `git` on a new machine, the first thing to do is to let `git` know about your name and e-mail address:

```
git config --global user.name My Name
git config --global user.email me@mydomain.net
```

Such information will be stored in commits. It is important to configure it properly when the time comes to generate and send patches, in particular.

Cloning the mainline Linux tree

To begin working with the Linux kernel sources, we need to clone its reference git tree, the one managed by Linus Torvalds.

The trouble is you have to download about 1.5 GB of data!

If you are running this command from home, or if you have very fast access to the Internet at work (and if you are not 256 participants in the training room), you can do it directly by connecting to `http://git.kernel.org`:

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

or if the network port for `git` is blocked by the corporate firewall, you can use the `http` protocol as a less efficient fallback:

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

If Internet access is not fast enough and if multiple people have to share it, your instructor will give you a USB flash drive with a `tar.xz` archive of a recently cloned Linux source tree.

You will just have to extract this archive in the current directory, and then pull the most recent changes over the network:

```
tar Jxf linux-git.tar.xz
cd linux
git pull
```


Of course, if you directly ran `git clone`, you won't have run `git pull` today. You may run `git pull` every morning though, or at least every time you need an update of the upstream source tree.

Accessing stable releases

Having the Linux kernel development sources is great, but when you are creating products, you prefer to avoid working with target that moves every day.

That's why we need to use the *stable* releases of the Linux kernel.

Fortunately, with `git`, you won't have to clone an entire source tree again. All you need to do is add a reference to a *remote* tree, and fetch only the commits which are specific to that remote tree.

```
cd ~/felabs/linux/src/linux/
git remote add stable git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
git fetch stable
```

This still represents about 80 MB worth of git objects to download, as the time of this writing (as of 3.11.1).

We will choose a particular stable version in the next labs.

Now, let's continue the lectures. This will leave time for the commands that you typed to complete their execution (if needed).

Kernel source code

Objective: Get familiar with the kernel source code

After this lab, you will be able to:

- Create a branch based on a remote tree to explore a particular stable kernel version (from the `stable` kernel tree).
- Explore the sources in search for files, function headers or other kinds of information...
- Browse the kernel sources with tools like `cscope` and LXR.

Choose a particular stable version

Let's work with a particular stable version of the Linux kernel. It would have been more logical to do this in the previous lab, but we wanted to get back to lectures while the `fetch` command was running.

First, let's get the list of branches on our `stable` remote tree:

```
cd ~/felabs/linux/src/linux
git branch -a
```

As we want to work with the Linux 3.11 stable branch, the remote branch we are interested in is `remotes/stable/linux-3.11.y`.

First, open the `Makefile` file just to check the Linux kernel version that you currently have.

Now, let's create a local branch starting from that remote branch:

```
git checkout -b 3.11.y stable/linux-3.11.y
```

Open `Makefile` again and make sure you now have a 3.11.y version.

Exploring the sources manually

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 MVNETA network driver is.
3. Find the declaration of the `platform_device_register()` function.

Tip: if you need the `grep` command, we advise you to use `git grep`. This command is similar, but much faster, doing the search only on the files managed by git (ignoring git internal files and generated files).

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` instead.

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

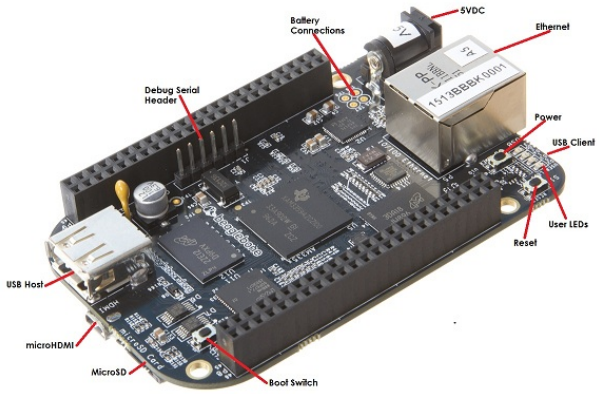
Board setup

Objective: setup communication with the board and configure the bootloader.

After this lab, you will be able to:

- Access the board through its serial line
- Create a bootable micro-SD card to boot the board with
- Configure the U-boot bootloader and a tftp server on your workstation to download files through tftp

Getting familiar with the board



Take some time to read about the board features and connectors on http://circuitco.com/support/index.php?title=BeagleBoneBlack. The above image was taken from this page.

Then download the board System Reference Manual found at https://github.com/CircuitCo/BeagleBone-Black/blob/master/BBB_SRM.pdf?raw=true [1]. This is the ultimate reference about the board, giving all the details about the design of the board and the components which were chosen. You don't have to start reading this document now but you will need it during the practical labs.

Last but not least, download the Technical Reference Manual (TRM) for the TI AM3359 SoC, available on http://www.ti.com/product/am3359. This document is more than 4700 pages big (20 MB)! You will need it too during the practical labs.

Don't hesitate to share your questions with the instructor.

[1] There is a link to this manual on the above page

Setting up serial communication with the board

The Beaglebone serial connector is exported on the 6 pins close to one of the 48 pins headers. Using your special USB to Serial adaptor provided by your instructor, connect the ground wire (blue) to the pin closest to the power supply connector (let's call it pin 1), and the `TX` (red) and `RX` (green) wires to the pins 4 (`RX`) and 5 (`TX`). [2]

You always should make sure that you connect the `TX` pin of the cable to the `RX` pin of the board, and vice versa, whatever the board and cables that you use.

Once the USB to Serial connector is plugged in, a new serial port should appear: `/dev/ttyUSB0`. You can also see this device appear by looking at the output of `dmesg`.

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

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

If you run `ls -l /dev/ttyUSB0`, you can also see that only `root` and users belonging to the `dialout` group have read and write access to this file. Therefore, you need to add your user to the `dialout` group:

```
sudo adduser $USER dialout
```

You now need to log out and log in again to make the new group visible everywhere.

Now, you can 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]`.

There should be nothing on the serial line so far, as the board is not powered up yet.

Before booting your board, make sure that there is no micro-SD card in the corresponding slot.

It is now time to power up your board by plugging in the mini-USB cable supplied by your instructor (with your PC or a USB power supply at the other end of the cable).

See what messages you get on the serial line. You should see Linux boot on the serial line.

Bootloader interaction

Reset your board. Press a key in the `picocom` terminal to stop the U-boot countdown. You should then see the U-Boot prompt:

```
U-Boot>
```

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

Setting up Ethernet communication

The next step is to configure U-boot and your workstation to let your board download files, such as the kernel image and Device Tree Binary (DTB), using the TFTP protocol through an Ethernet cable.

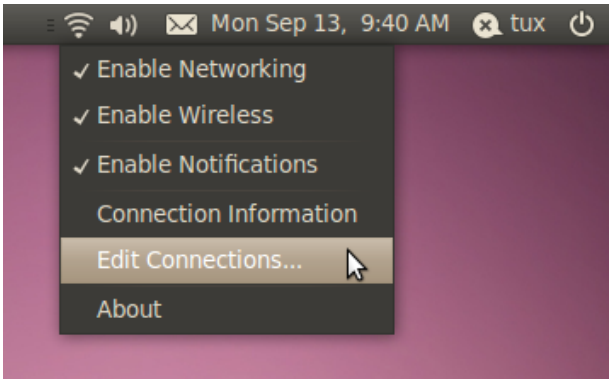
To start with, install a TFTP server on your development workstation:

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

[2] See https://www.olimex.com/Products/Components/Cables/USB-Serial-Cable/USB-Serial-Cable-F/ for details about the USB to Serial adaptor that we are using.

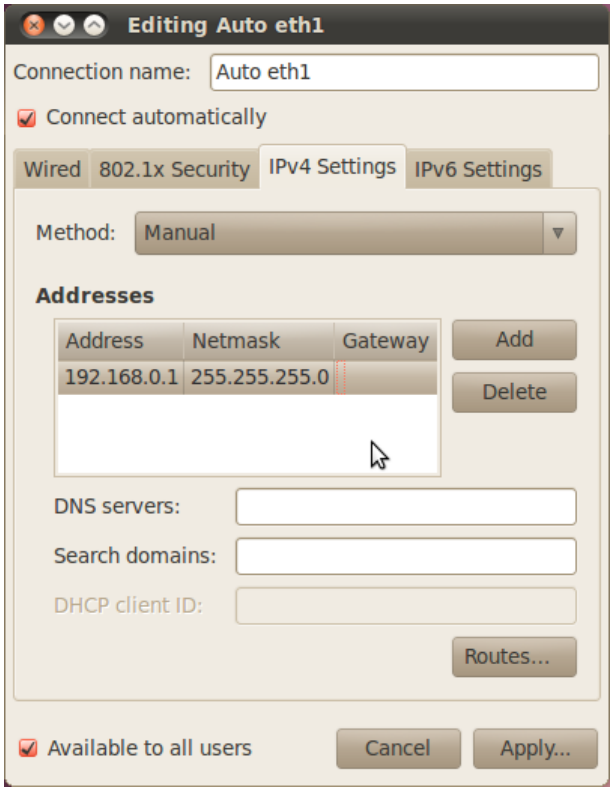
With a network cable, connect the Ethernet port of your board to the one of your computer. If your computer already has a wired connection to the network, your instructor will provide you with a USB Ethernet adapter. A new network interface, probably `eth1` or `eth2`, should appear on your Linux system.

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:

In the `IPv4 Settings` tab, press the `Add` button and make the interface use a static IP address, like `192.168.0.1` (of course, make sure that this address belongs to a separate network segment from the one of the main company network). You will also need to specify the local network mask (*netmask*, often `255.255.255.0`). You can keep the `Gateway` field empty (don't click put the cursor inside the corresponding text box, otherwise it will ask for a legal value) or set it to `0.0.0.0`:



Now, it's time to configure networking on U-Boot's side.

Back to the U-Boot command line, set the below environment variables:

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

Save these settings to the eMMC storage on the board: [3]

```
saveenv
```

You can then test the TFTP connection. First, put a small text file in `/var/lib/tftpboot`. Then, from U-Boot, do:

```
tftp 0x81000000 textfile.txt
```

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

```
/etc/init.d/tftpd-hpa restart
```

The `tftp` command should command should have downloaded the `textfile.txt` file from your development workstation into the board's memory at location `0x81000000` (this location is part of the board DRAM). You can verify that the download was successful by dumping the contents of the memory:

```
md 0x81000000
```

We are now ready to load and boot a Linux kernel!

[3] The U-boot environment settings are stored in some free space between the master boot record (512 bytes, containing the partition tables and other stuff), and the beginning of the first partition (often at `32256`). This is why you won't find any related file in the first partition of the eMMC storage.

Kernel compiling and booting

Objective: compile and boot a kernel for your board, booting on a directory on your workstation shared by NFS.

After this lab, you will be able to:

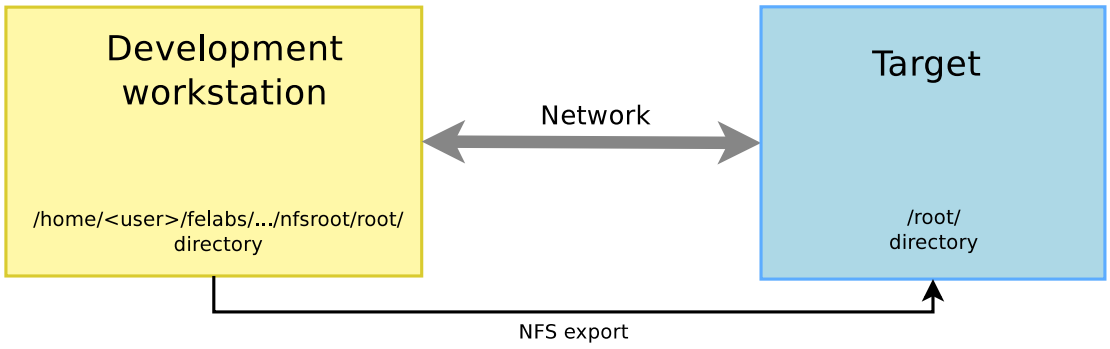
- Cross-compile a kernel for the ARM platform
- Boot this kernel on an NFS root filesystem, which is somewhere on your development workstation[4]

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

Go to the `$HOME/felabs/linux/src/linux` directory.

Install packages needed for configuring, compiling and booting a kernel for your board:

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

`libqt4-dev` and `g++` are needed for `make xconfig`. `u-boot-tools` is needed to build the `uImage` file for U-boot (`mkimage` utility).

[4] NFS root filesystems are particularly useful to compile modules on your host, and make them directly visible on the target. You no longer have to update the root filesystem by hand and transfer it to the target (requiring a shutdown and reboot).

Apply a kernel patch for BeagleBone Black

At the time of this writing, the mainline Linux 3.11 kernel is missing a specific Device Tree Source (DTS) file for the BeagleBone Black. Without it, you could use the DTS for the original Beagle Bone, but it would blow up the HDMI transceiver after a dozen boots [5].

We are providing a patch to add such a DTS file. Let's create a branch, starting from our `3.11.y` branch, and including this patch.

First, make sure you are in the `3.11.y` branch:

```
git branch
```

Then, let's create the new branch and apply the patch for BeagleBone Black:

```
git checkout -b 3.11.y-bbb
git am ../patches/0001-ARM-OMAP2-*.patch
```

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 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
```

Kernel configuration

Configure this kernel with the ready-made configuration for boards in the OMAP2 and later family which the AM335x found in the BeagleBone belongs to. Don't forget to set the `ARCH` and `CROSS_COMPILE` definitions for the `arm` platform and to use your cross-compiler.

Make sure that this configuration has `CONFIG_ROOT_NFS=y` (support booting on an NFS exported root directory).

Compile your kernel and generate the `uImage` kernel image that U-boot needs (the U-boot bootloader needs the kernel `zImage` file to be encapsulated in a special container and the kernel `Makefile` can generate this container for you by running the `mkimage` tool found in the `uboot-mkimage` package).

This file will contain, among other things, the load address of the kernel. Nowadays, the kernel can boot on several platforms, each with different load addresses, that makes the use of `uImage` not very convenient. So, if the default load address doesn't work for you, you'll need to specify it during the generation of `uImage` using the `LOADADDR` environment variable.

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

Now also generate the Device Tree Binaries (DTBs):

```
make dtbs
```

Now, copy the `uImage` and `am335x-boneblack.dtb` files to the TFTP server home directory (`/var/lib/tftpboot`).

[5] See http://article.gmane.org/gmane.linux.kernel.stable/63648 for details.

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)
```

Then, restart the NFS server:

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

If there is any error message, this usually means that there was a syntax error in the `/etc/exports` file. Don't proceed until these errors disappear.

Boot the system

First, boot the board to the U-Boot prompt. Before booting the kernel, we need to tell it which console to use and that the root filesystem should be mounted over NFS, by setting some kernel parameters.

Do this by setting U-boot's `bootargs` environment variable (all in just one line, pay attention to the `O` character, like "OMAP", in `ttyO0`):

```
setenv bootargs root=/dev/nfs ip=192.168.0.100 console=ttyO0
  nfsroot=192.168.0.1:/home/<user>/felabs/linux/modules/nfsroot
saveenv
```

Of course, you need to adapt the IP addresses to your exact network setup. Now, download the kernel image through `tftp`:

```
tftp 0x81000000 uImage
```

You'll also need to download the device tree blob:

```
tftp 0x82000000 am335x-boneblack.dtb
```

Now, boot your kernel:

```
bootm 0x81000000 - 0x82000000
```

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`.

Automate the boot process

To avoid typing the same U-boot commands over and over again each time you power on or reset your board, you can use U-Boot's `bootcmd` environment variable:

```
setenv bootcmd 'tftp 0x81000000 uImage; tftp 0x82000000 am335x-boneblack.dtb; bootm 0x81000000 - 0x82000000'
saveenv
```

Don't hesitate to change it according to your exact needs.

We could also copy the `uImage` file to the eMMC flash and avoid downloading it over and over again. However, detailed bootloader usae is outside of the scope of this course. See our Embedded Linux system development course and its on-line materials for details.

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

Go to the `~/felabs/linux/modules/nfsroot/root/hello` directory. Boot your board if needed.

Writing a module

Look at the contents of the current directory. All the files you generate there will also be visible from the target. That's great to load modules!

Add C code to the `hello_version.c` file, to implement a module which displays this kind of message when loaded:

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

... and displays a goodbye message when unloaded.

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

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. Compile your module.

Testing your module

Load your new module file on the target. 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.pl -h` command in the kernel sources, to find which options are available. Now, run:

```
~/felabs/linux/src/linux/scripts/checkpatch.pl --file --no-tree 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 `indent` command:

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

Caution: don't run `indent` when you modify source files created by other people. Otherwise, people won't be able to distinguish your own changes from the ones made by `indent`.

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` module to the kernel sources

As we are going to make changes to the kernel sources, first create a special branch for such changes:

```
git checkout 3.11.y-bbb
git checkout -b hello
```


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.

Run the one of the kernel configuration interfaces and check that it shows your new driver 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, commit your changes in the current branch (try to choose an appropriate commit message):

```
cd ~/felabs/linux/src/linux
git add -A
git commit -as
```

- `git add -A` adds (or removes) files to the next commit (except for files explicitly ignored, such as generated ones). Another, perhaps safer way to do this without taking the risk to add unwanted files, is to run `git status` and explicitly run `git add` on each of the files that you want to add to the next commit.
- `git commit -a` creates a commit with all modified files (at least the ones tracked by the repository) since the previous commit.
- `git commit -s` adds a `Signed-off-by:` line to the commit message. All contributions to the Linux kernel must have such a line.

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 mainline kernel.

Creating a patch with `git` is extremely easy! You just generate it from the commits between your branch and another branch, usually the one you started from:

```
git format-patch 3.11.y-bbb
```

Have a look at the generated file. You can see that its name reused the commit message.

If you want to change the last commit message at this stage, you can run:

```
git commit --amend
```

And run `git format-patch` again.

Device Model - I2C device

Objective: declare an I2C device and basic driver hooks called when this device is detected

Throughout the upcoming labs, we will implement a driver for an I2C device, which offers the functionality of an I2C nunchuk.

After this lab, you will be able to:

- Add an I2C device to a device tree
- Implement basic `probe()` and `remove()` driver functions and make sure that they are called when there is a device/driver match.
- Find your driver and device in `/sys`

Setup

Go to the `~/felabs/linux/src/linux` directory. Check out the `3.11.y-bbb` branch.

Now create a new `nunchuk` branch starting from the `3.11.y-bbb` branch, for your upcoming work on the nunchuk driver.

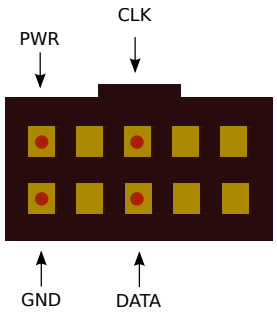
Connecting the nunchuk

Take the nunchuk device provided by your instructor.

We will connect it to the second I2C port of the CPU (`i2c1`), which pins are available on the `P9` connector.

Download a useful document sharing useful details about the nunchuk and its connector: http://web.engr.oregonstate.edu/~sullivae/ece375/pdf/nunchuk.pdf

Now we can identify the 4 pins of the nunchuk connector:

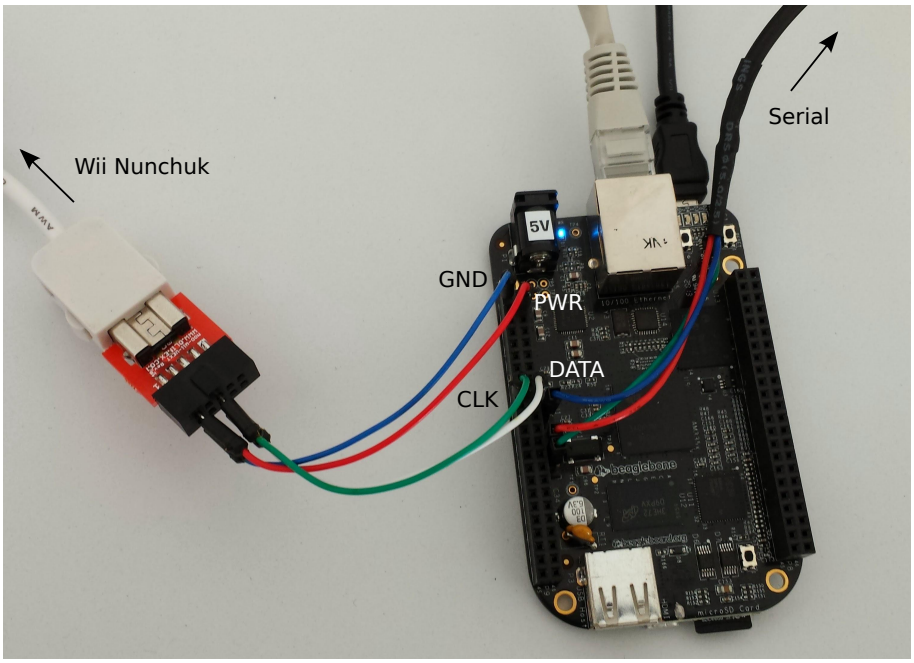


Nunchuk i2c pinout
(UEXT connector from Olimex)

Open the System Reference Manual that you downloaded earlier, and look for "connector P9" in the table of contents, and then follow the link to the corresponding section. Look at the table listing the pinout of the P9 connector.

Now connect the nunchuk pins:

- The `GND` pin to P9 pins 1 or 2 (`GND`)
- The `PWR` pin to P9 pins 3 or 4 (`DC_3.3V`)
- The `CLK` pin to P9 pin 17 (`I2C1_SCL`)
- The `DATA` pin to P9 pin 18 (`I2C1_SDA`)



Update the board device tree

To let the Linux kernel handle a new device, the first thing is to add a description for it in the board device tree.

Do this by editing the `arch/arm/boot/dts/am335x-bone-common.dtsi` file describing all the buses and devices. You will need to follow the examples given in the lectures.

1. Add a node declaring a second I2C bus (`i2c1`), functioning at 100 KHz. As for `i2c0`, you will need to declare the base address of its registers. Open the processor datasheet and find this address [6].

2. As a child node to this second bus, declare the `nunchuk` device, choosing `nintendo,nunchuk` for its `compatible` property. You find the I2C slave address of the nunchuk on the nunckuk document that we have used earlier [7].

Once this is done, recompile your DTB and copy the updated version to the tftp server home directory.

[6] Tip: you can look-up the `i2c0` base address which you already know from the existing Device Tree. The base address for `i2c1` won't be far away. If you do so, ignore the `pinctrl` properties for now.

[7] This I2C slave addressed is enforced by the device itself. You can't change it.

Implement a basic I2C driver for the nunchuk

It is now time to start writing the first building blocks of the I2C driver for our nunchuk.

In a new terminal, go to `~/felabs/linux/modules/nfsroot/root/nunchuk/`. This directory contains a Makefile and an almost empty `nunchuk.c` file.

Now, you can compile your out-of-tree module by running `make`. As the current directory is part of the NFS root that the board boots on, the generated `.ko` file will immediately be visible on the board too.

Relying on explanations given during the lectures, fill the `nunchuk.c` file to implement:

- `probe()` and `remove()` functions that will be called when a nunchuk is found. For the moment, just put a call to `pr_info()` inside to confirm that these function are called.
- Initialize an `i2c_driver` structure, and register the i2c driver using it. Make sure that you use a `compatible` property that matches the one in the Device Tree.

You can now compile your module and reboot your board, to boot with the updated DTB.

Driver tests

You can now load the `/root/nunchuk/nunchuk.ko` file. You need to check that the `probe()` function gets called then, and that the `remove()` function gets called too when you remove the module.

Once your new Device Tree and module work as expected, commit your DT changes in your Linux tree:

```
git commit -sa
```

Exploring /sys

Take a little time to explore `/sys`:

- Find the representation of your driver. That's a way of finding the matching devices.
- Find the representation of your device, containing its name. You will find a link to the driver too.

Using the I2C bus

Objective: make the I2C bus work and use it to implement communication with the Nunchuk device

After this lab, you will be able to:

- Declare pinctrl settings
- Access I2C device registers through the bus

Setup

Stay in the `~/felabs/linux/src/linux` directory for kernel and DTB compiling (stay in the `nunchuk` branch), and in `~/felabs/linux/modules/nfsroot/root/nunchuk` for module compiling (use two different terminals).

Add pinctrl properties to the Device Tree

As you found in the previous lab, we now managed to have our nunchuk device enumerated on the `i2c1` bus.

However, to access the bus data and clock signals, we need to configure the pin muxing of the SoC.

If you go back to the BeagleBone Black System Reference Manual, in the *Connector P9* section, you can see that the pins `17` and `18` that we are using correspond to pins `A16` and `B16` of the AM335 SoC. You can also see that such pins need to be configured as `MODE2` to get the functionality that we need (`I2C1_SCL` and `I2C1_SDA`).

Now look at the Device Tree for the AM335x EVM board (`arch/arm/boot/dts/am335x-evm.dts`). It's using `i2c1` too.

Edit the `arch/arm/boot/dts/am335x-bone-common.dtsi` file and add what's needed to enable pin muxing for `i2c1`. Don't hesitate to go back to the lectures to understand what to do!

Rebuild and update your DTB, and eventually reboot the board.

We will use the `i2cdetect` command to make sure that everything works fine for `i2c1`:

```
# i2cdetect -l
i2c-0 i2c         OMAP I2C adapter                I2C adapter
i2c-1 i2c         OMAP I2C adapter                I2C adapter
```

```
# i2cdetect -F 1
Functionalities implemented by /dev/i2c-1:
I2C                              yes
SMBus Quick Command              no
SMBus Send Byte                  yes
SMBus Receive Byte               yes
SMBus Write Byte                 yes
```



```
SMBus Read Byte                   yes
SMBus Write Word                  yes
SMBus Read Word                   yes
SMBus Process Call                yes
SMBus Block Write                 yes
SMBus Block Read                  no
SMBus Block Process Call          no
SMBus PEC                         yes
I2C Block Write                   yes
I2C Block Read                    yes
```

You can see that the *SMBus Quick Commands* are not available on this driver, yet `i2cdetect` uses them by default to scan the i2c bus. You can use `i2cdetect -r` to use the usual set of i2c commands, and be able to detect the devices on your bus.

To test if everything works fine, run `i2cdetect -r 1`. This will scan the `i2c1` bus for devices. You should see a device at the address `0x52`. This is your nunchuk.

If everything works as expected, commit your Device Tree changes. This will be required switch to another branch later:

```
git commit -as
```

- `git commit -a` adds all the files already known to `git` to the commit.
- `git commit -s` adds a `Signed-off-by` line (required for all contributions to the Linux kernel).

Device initialization

The next step is to read the state of the nunchuk registers, to find out whether buttons are pressed or not, for example.

Before being able to read nunchuk registers, the first thing to do is to send initialization commands to it. That's also a nice way of making sure i2c communication works as expected.

In the probe routine (run every time a matching device is found):

1. Using the I2C raw API (see the slides), send two bytes to the device: `0xf0` and `0x55` [8]. Make sure you check the return value of the function you're using. This could reveal communication issues. Using LXR, find examples of how to handle failures properly using the same function.
2. Let the CPU wait for 1 ms by using the `udelay()` routine. You may need to use LXR again to find the right C headers to include.
3. In the same way, send the `0xfb` and `0x00` bytes now. This completes the nunchuk initialization.

Recompile and load the driver, and make sure you have no communication errors.

[8] There are two ways of communicating with a wiimote extension. The first known way was with data encryption by writing the encryption byte 0x00 to 0x40. With this way, you have to decrypt each bytes you read from the nunchuk (not so hard but something you have to do). Unfortunately, using 0x00 as the encryption byte is not working on third party nunchuks so you have to set up unencrypted communication by writing 0x55 to 0xF0, then writing 0x00 to 0xFB. This is working across all brands of nunchuks (including Nintendo ones).

Read nunchuk registers

The nunchuk exhibits a rather weird behaviour: it seems that it update the state of its internal registers only when they have been read.

As a consequence, we will need to read the registers twice!

To keep the code simple and readable, let's create a `nunchuk_read_registers` function to do this. In this function:

1. Start by putting a `10 ms` delay by calling the `mdelay()` routine. That's needed to add time between the previous i2c operation and the next one.
2. Write `0x00` to the bus. That will allow us to read the device registers.
3. Add another `10 ms` delay.
4. Read 6 bytes from the device, still using the I2C raw API. Check the return value as usual.

Reading the state of the nunchuk buttons

Back to the `probe()` function, call your new function twice.

After the second call, compute the states of the `Z` and `C` buttons, which can be found in the sixth byte that you read.

As explained on http://web.engr.oregonstate.edu/~sullivae/ece375/pdf/nunchuk.pdf:

- `bit 0 == 0` means that `Z` is pressed.
- `bit 0 == 1` means that `Z` is released.
- `bit 1 == 0` means that `C` is pressed.
- `bit 1 == 1` means that `C` is released.

Using boolean operators, write code that initializes a `zpressed` integer variable, which value is `1` when the `Z` button is pressed, and `0` otherwise. Create a similar `cpressed` variable for the `C` button.[9]

The last thing is to test the states of these new variables at the end of the `probe()` function, and log a message to the console when one of the buttons is pressed.

Testing

Compile your module, and reload it. No button presses should be detected. Remove your module.

Now hold the `Z` and reload and remove your module again:

```
insmod /root/nunchuk/nunchuk.ko; rmmod nunchuk
```

You should now see the message confirming that the driver found out that the `Z` button was held.

Do the same over and over again with various button states.

[9] You may use the `BIT()` macro, which will make your life easier. See LXR for details.

At this stage, we just made sure that we could read the state of the device registers through the I2C bus. Of course, loading and removing the module every time is not an acceptable way of accessing such data. We will give the driver a proper *input* interface in the next slides.

Input interface

Objective: make the I2C device available to userspace using the input subsystem.

After this lab, you will be able to:

- Expose device events to userspace through an input interface, using the kernel based polling API for input devices (kernel space perspective)
- Handle registration and allocation failures in a clean way.
- Get more familiar with the usage of the input interface (user space perspective)

Add polled input device support to the kernel

The nunchuk doesn't have interrupts to notify the I2C master that its state has changed. Therefore, the only way to access device data and detect changes is to regularly poll its registers, using the input polling API described in the lectures.

Rebuild your kernel with static support for polled input device support (`CONFIG_INPUT_POLLDEV=y`). With the default configuration, this feature is available as a module, which is less convenient.

Update and reboot your kernel.

Register an input interface

The first thing to do is to add an input device to the system. Here are the steps to do it:

- Declare a pointer to an `input_polled_dev` structure in the `probe` routine. You can call it `polled_input`. You can't use a global variable because your driver needs to be able to support multiple devices.
- Allocate such a structure in the same function, using the `input_allocate_polled_device()` function.
- Also declare a pointer to an `input_dev` structure. You can call it `input`. We won't need to allocate it, because it is already part of the `input_polled_dev` structure, and allocated at the same time. We will use this as a shortcut to keep the code simple.
- Still in the `probe()` function, add the input device to the system by calling `input_register_polled_device();`

At this stage, first make sure that your module compiles well (add missing headers if needed).

Handling probe failures

In the code that you created, make sure that you handle failure situations properly.

- Of course, test return values values properly and log the causes of errors.

- If the call to `input_register_polled_device()` fails, you must also free the `input_polled_dev` structure before returning an error. If you don't do that, you will create memory leaks in the kernel. In the general case, failure to release things that have been allocated or registered before can prevent you from reloading a module.

To implement this correctly without duplicating or creating ugly code, it's recommended to use `goto` statements.

See *Chapter 7: Centralized exiting of functions* in `Documentation/CodingStyle` for useful guidelines and an example. Implement this in your driver.

Implement the remove() function

We now need to release the resources allocated and registered in the `probe()` routine.

However, this is not trivial in the way we implemented the `probe()` routine. As we have to support multiple devices, we chose not to use global variables, and as a consequence, we can't use such global variables to release the corresponding resources.

This raises a very important aspect of the device model: the need to keep pointers between *physical* devices (devices as handled by the physical bus, I2C in our case) and *logical* devices (devices handled by subsystems, like the input subsystem in our case).

This way, when the `remove()` routine is called (typically if the bus detects the removal of a device), we can find out which logical device to unregister. Conversely, when we have an event on the logical side (such as opening or closing an input device for the first time), we can find out which i2c slave this corresponds to, to do the specific things with the hardware.

This need is typically implemented by creating a *private* data structure to manage our device and implement such pointers between the physical and logical worlds.

Add the below definition to your code:

```
struct nunchuk_dev {
        struct input_polled_dev *polled_input;
        struct i2c_client *i2c_client;
};
```

Now, in your `probe()` routine, declare an instance of this structure:

```
        struct nunchuk_dev *nunchuk;
```

Then allocate one such instead for each new device:

```
        nunchuk = kzalloc(sizeof(struct nunchuk_dev), GFP_KERNEL);
        if (!nunchuk) {
                dev_err(&client->dev, "Failed to allocate memory\n");
                return -ENOMEM;
        }
```

Note that we haven't seen kernel memory allocator routines and flags yet. We haven't explained the `dev_*` logging routines yet either (they are basically used to tell which device a given log message is associated to). For the moment, just use the above code. You will get the details later.

You will actually have to modify your code and the above lines to call `kfree(nunchuk)` if one of the subsequent registration and allocation routines fail.

Now implement the pointers:


```
nunchuk->i2c_client = client;
nunchuk->polled_input = polled_input;
polled_input->private = nunchuk;
i2c_set_clientdata(client, nunchuk);
input = polled_input->input;
input->dev.parent = &client->dev;
```

Make sure you add this code before registering the input device. You don't want to enable a device with incomplete information or when it is not completely yet (there could be race conditions).

With all this in place, you now have everything you need to implement the `remove()` routine. Look at the I2C and input slides for examples, or directly find your examples in the Linux kernel code!

Recompile your module, and load it and remove it multiple times, to make sure that everything is properly registered and unregistered.

Add proper input device registration information

We actually need to add more information to the input structure before registering it. That's why we are getting the below warnings:

```
input: Unspecified device as /devices/virtual/input/input0
```

Add the below lines of code (still before device registration, of course):

```
input->name = "Wii Nunchuk";
input->id.bustype = BUS_I2C;

set_bit(EV_KEY, input->evbit);
set_bit(BTN_C, input->keybit);
set_bit(BTN_Z, input->keybit);
```

Recompile and reload your driver. You should now see in the kernel log that the `Unspecified device` type is replaced by codeWii Nunchuck and that the physical path of the device is reported too.

Implement the polling routine

It's time to implement the routine which will poll the nunchuk registers at a regular interval.

Create a `nunchuck_poll()` function with the right prototype (find it by looking at the definition of the `input_polled_dev` structure.

First, add lines retrieving the I2C physical device from the `input_polled_dev` structure. That's where you will need your private `nunchuk` structure.

Now that you have a handle on the I2C physical device, you can move the code reading the nunchuk registers to this function. You can remove the double reading of the device state, as the polling function will make periodic reads anyway [10].

At the end of the polling routine, the last thing to do is post the events and notify the `input` core:

[10] During the move, you will have to handle communication errors in a slightly different way, as the `nunchuk_poll()` routine has a `void` type. When the function reading registers fails, you can use a `return;` statement instead of `return value;`


```
input_event(nunchuk->polled_input->input,
            EV_KEY, BTN_Z, zpressed);
input_event(nunchuk->polled_input->input,
            EV_KEY, BTN_C, cpressed);

input_sync(nunchuk->polled_input->input);
```

Now, back to the `probe()` function, the last thing to do is to declare the new polling function (see the slides if you forgot about the details) and specify a polling interval of 50 ms.

You can now make sure that your code compiles and loads successfully.

Testing your input interface

Testing an input device is easy with the `evtest` application that is included in the root filesystem. Just run:

```
evtest
```

The application will show you all the available input devices, and will let you choose the one you are interested in (make sure you type a choice, `0` by default, and do not just type `[Enter]`). You can also type `evtest /dev/input/event0` right away.

Press the various buttons and see that the corresponding events are reported by `evtest`.

Going further

If you complete your lab before the others, you can add support for the nunchuk joystick coordinates.

Another thing you can do then is add support for the nunchuk accelerometer coordinates.

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 additional CPU serial ports available on the BeagleBone, and to read data from them.

After this lab, you will be able to:

- Add UART devices to the board device tree
- Access I/O registers to control the device and send first characters to it.

Setup

Go to your kernel source directory.

Create a new branch for this new series of labs. Since this new stuff is independent from the nunchuk changes, it's best to create a separate branch!

```
git checkout 3.11.y-bbb
git checkout  -b uart
```

Add UART devices

Before developing a driver for additional UARTS on the board, we need to add the corresponding descriptions to the board Device Tree.

First, open the board reference manual and find the connectors and pinmux modes for UART2 and UART4.

Using an new USB-serial cable with male connectors, provided by your instructor, connect your PC to UART2. The wire colors are the same as for the cable that you're using for the console:

- The blue wire should be connected `GND`
- The red wire (`TX`) should be connected to the board's `RX` pin
- The green wire (`RX`) should be connected to the board's `TX` pin

You can (or even should) show your connections to the instructor to make sure that you haven't swapped the `RX` and `TX` pins.

Now, open the `arch/arm/boot/dts/am335x-bone-common.dtsi` file and create declarations for UART2 and UART4 in the pin muxing section:

```
    /* Pins 21 (TX) and 22 (RX) of connector P9 */
    uart2_pins: uart2_pins {
        pinctrl-single,pins = <
            0x154 (PIN_OUTPUT_PULLDOWN | MUX_MODE1) /* spi0_d0.uart2_tx, MODE 1 */
            0x150 (PIN_INPUT_PULLUP | MUX_MODE1) /* spi0_sclk.uart2_rx, MODE 1 */
        >;
    };
```



```
/* Pins 11 (RX) and 13 (TX) of connector P9 */
uart4_pins: uart4_pins {
    pinctrl-single,pins = <
        0x74 (PIN_OUTPUT_PULLDOWN | MUX_MODE6) /* gpmc_wpn.uart4_tx, MODE 6 */
        0x70 (PIN_INPUT_PULLUP | MUX_MODE6) /* gpmc_wait0.uart4_rx, MODE 6 */
    >;
};
```

Then, declare the corresponding devices:

```
uartfe2: feserial@48024000 {
    compatible = "free-electrons,serial";
    /* Tell the OMAP hardware power management that the block
       must be enabled, otherwise it's switched off
       Caution: starting counting at 1, not 0 */
    ti,hwmods = "uart3";
    clock-frequency = <48000000>;
    reg = <0x48024000 0x2000>;
    interrupts = <74>;
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <&uart2_pins>;
};

uartfe4: feserial@481a8000 {
    compatible = "free-electrons,serial";
    ti,hwmods = "uart5";
    clock-frequency = <48000000>;
    reg = <0x481a8000 0x2000>;
    interrupts = <45>;
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <&uart4_pins>;
};
```

Note: we are calling these devices with `uartfe` instead of `uart` to avoid conflicts with declarations in `arch/arm/boot/dts/am33xx.dtsi`. The `uart` devices are meant to be used by the regular serial driver.

We will see how to use the device parameters in the driver code.

Rebuild and update your DTB.

Operate a platform device driver

Go to the `~/felabs/linux/modules/nfsroot/root/serial/` directory. You will find a `feserial.c` file already provides a platform driver skeleton.

Add the code needed to match the driver with the devices which you have just declared in the device tree.

Compile your module and load it on your target. Check the kernel log messages, that should confirm that the `probe()` routine was called [11].

[11] Don't be surprised if the `probe()` routine is actually called twice! That's because we have declared two devices.

Get base addresses from the device tree

We are going to read from memory mapped registers and read from them. The first thing we need is the base physical address for the each device.

Such information is precisely available in the Device Tree. You can extract it with the below code:

```
struct resource *res;
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
```

Add such code to your `probe()` routine, with proper error handling when `res == NULL`, and print the start address (`res->start`) to make sure that the address values that you get match the ones in the device tree.

You can remove the printing instruction as soon as the collected addresses are correct.

Create a device private structure

The next step is to start allocating and registering resources, which eventually will have to be freed and unregistered too.

In the same way as in the nunchuk lab, we now need to create a structure that that will hold device specific information and help keeping pointers between logical and physical devices.

As the first thing to store will be the base virtual address for each device (obtained through `ioremap()`), let's declare this structure as follows:

```
struct feserial_dev {
        void __iomem *regs;
};
```

The first thing to do is allocate such a structure at the beginning of the `probe()` routine. Let's do it with a `devm_` function. The advantage of such routines is that each allocation or registration is attached to a device structure. When a device or a module is removed, all such allocations or registrations are automatically undone. This allows to greatly simplify driver code.

So, add the below line to your code:

```
struct feserial_dev *dev;
...
dev = devm_kzalloc(&pdev->dev, sizeof(struct feserial_dev), GFP_KERNEL);
```

You can now get a virtual address for your device's base physical address, by calling:

```
        dev->regs = devm_request_and_ioremap(&pdev->dev, res);

        if (!dev->regs) {
                dev_err(&pdev->dev, "Cannot remap registers\n");
                return -ENOMEM;
        }
```

What's nice is that you won't ever have to release this resource, neither in the `remove()` routine, nor if there are failures in subsequent steps of the `probe()` routine.

Make sure that your updated driver compiles, loads and unloads well.

Even if we only connect a serial-to-USB dongle to one of them, both of them are ready to be used!

Device initialization

Now that we have a virtual address to access registers, we are ready to configure a few registers which will allow us to enable the UART devices. Of course, this will be done in the `probe()` routine.

Accessing device registers

As we will have multiple registers to read, create a `reg_read()` routine, returning an `unsigned int` value, and taking a `dev` pointer to an `feserial_dev` structure and an `offset` integer offset.

In this function, read from a 32 bits register at the base virtual address for the device plus the offset multiplied by 4[12], and return this value.

Create a similar `reg_write()` routine, writing an unsigned integer value at a given integer offset (don't forget to multiply it by 4) from the device base virtual address. The following code samples are using the `writel()` convention of passing the value first, then the offset. Your prototype should look like:

```
static void reg_write(struct feserial_dev *dev, int val, int off);
```

All the UART register offsets have standardized values, shared between several types of serial drivers (see `include/uapi/linux/serial_reg.h`). This explains why they are not completely ready to use and we have to multiply them by 4 for OMAP SoCs.

We are now ready to read and write registers!

Power management initialization

Add the below lines to the probe function:

```
pm_runtime_enable(&pdev->dev);
pm_runtime_get_sync(&pdev->dev);
```

And add the below line to the `remove()` routine:

```
pm_runtime_disable(&pdev->dev);
```

Line and baud rate configuration

After these lines, let's add code to initialize the line and configure the baud rate. This shows how to get a special property from the device tree, in this case `clock-frequency`:

```
/* Configure the baud rate to 115200 */

of_property_read_u32(pdev->dev.of_node, "clock-frequency",
                     &uartclk);
baud_divisor = uartclk / 16 / 115200;
reg_write(dev, 0x07, UART_OMAP_MDR1);
reg_write(dev, 0x00, UART_LCR);
reg_write(dev, UART_LCR_DLAB, UART_LCR);
```

[12] You have to multiply by 4 because we are using a void* pointer so pointer arithmetic will use the offset as bytes whereas we are using 32 bits registers.

```
reg_write(dev, baud_divisor & 0xff, UART_DLL);
reg_write(dev, (baud_divisor >> 8) & 0xff, UART_DLM);
reg_write(dev, UART_LCR_WLEN8, UART_LCR);
```

Declare `baud_divisor` and `uartclk` as `unsigned int`.

Soft reset

The last thing to do is to request a software reset:

```
/* Soft reset */
reg_write(dev, UART_FCR_CLEAR_RCVR | UART_FCR_CLEAR_XMIT, UART_FCR);
reg_write(dev, 0x00, UART_OMAP_MDR1);
```

We are now ready to transmit characters over the serial ports!

If you have a bit of spare time, you can look at section 19 of the AM335x TRM for details about how to use the UART ports, to understand better what we are doing here.

Standalone write routine

Implement a C routine taking a pointer to an `feserial_dev` structure and one character as parameters, and writing this character to the serial port, using the following steps:

1. Wait until the `UART_LSR_THRE` bit gets set in the `UART_LSR` register. You can busy-wait for this condition to happen. In the busy-wait loop, you can call the `cpu_relax()` kernel function to ensure the compiler won't optimise away this loop.
2. Write the character to the `UART_TX` register.

Add a call to this routine from your module `probe()` function, and recompile your module.

Open a new `picocom` instance on your new serial port (not the serial console):

```
picocom -b 115200 /dev/ttyUSB1
```

Load your module on the target. You should see the corresponding character in the new `picocom` installed, showing what was written to UART2.

You can also check that you also get the same character on UART4 (just connect to the UART4 pins instead of the UART2 ones).

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 registered, either at module exit time, or after a failure during the module `probe()` function. Check and fix your module code if you have such problems.

Output-only misc driver

Objective: implement the write part of a misc *driver*

After this lab, you will be able to:

- Write a simple misc driver, allowing to write data to the serial ports of your Beaglebone.
- 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.

You must have completed the previous lab to work on this one.

Misc driver registration

In the same way we added an input interface to our Nunchuk driver, it is now time to give an interface to our serial driver. As our needs are simple, we won't use the *Serial framework* provided by the Linux kernel, but will use the *Misc framework* to implement a simple character driver.

Let's start by adding the infrastructure to register a *misc* driver.

The first thing to do is to create:

- An `feserial_write()` write file operation stub. See the slides or the code for the prototype to use. Just place a `return -EINVAL;` statement in the function body so far, to signal that there something wrong with this function so far.
- Similarly, an `feserial_read()` read file operation stub.
- A `file_operations` structure declaring these file operations.

The next step is to create a `miscdevice` structure and initialize it. However, we are facing the same usual constraint to handle multiple devices. Like in the Nunchuk driver, we have to add such a structure to our device specific private data structure:

```
struct feserial_dev {
        struct miscdevice miscdev;
        void __iomem *regs;
};
```

To be able to access our private data structure in other parts of the driver, you need to attach it to the `pdev` structure using the `platform_set_drvdata()` function. Look for examples in the source code to find out how to do it.

Now, at the end of the `probe()` routine, when the device is fully ready to work, you can now initialize the `miscdevice` structure for each found device:

- To get an automatically assigned minor number
- To specify a name for the device file in *devtmpfs*. We propose to use `kasprintf(GFP_KERNEL, "feserial-\%x", res->start)`. `kasprintf()` allocates a buffer and

runs `ksprintf()` to fill its contents. Don't forget to call `kfree()` on this buffer in the `remove()` function!

- To pass the file operations structure that you defined.

See the lectures for details if needed!

The last things to do (at least to have a *misc* driver, even if its file operations are not ready yet), are to add the registration and deregistration routines. That's typically the time when you will need to access the `feserial_dev` structure for each device from the `pdev` structure passed to the `remove()` routine.

Make sure that your driver compiles and loads well, and that you now see two new device files in `/dev`.

At this stage, make sure you can load and unload the driver multiple times. This should reveal registration and deregistration issues if there are any.

Apply a kernel patch

The next step is to implement the `write()` routine. However, we will need to access our `feserial_dev` structure from inside that routine.

At the moment, it is necessary to implement an `open` file operation to attach a private structure to an open device file. This is a bit ugly as we would have nothing special to do in such a function.

Let's apply a patch that addresses this issue: [13]

- Go back to the Linux source directory. Make sure you are still in the `uart` branch (type `git branch`).
- Run `git status` to check whether you have uncommitted changes. Commit these if they correspond to useful changes (these should be your Device Tree edits).
- Apply the new patch using the following command: `git am ~/felabs/linux/src/patches/0001-char-misc*.patch`
- Rebuild and update your kernel image and reboot.

Implement the write() routine

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

The first thing to do is to retrieve the `feserial_dev` structure from the `miscdevice` structure, itself accessible through the `private_data` field of the open file structure (`file`).

At the time we registered our *misc* device, we didn't keep any pointeur to the `feserial_dev` structure. However, the `miscdevice` structure is accessible, and being a member of the `feserial_dev` structure, we can use a magic macro to compute the address of the parent structure:

```
struct feserial_dev *dev =
container_of(file->private_data, struct feserial_dev, miscdev);
```

[13] This patch has been submitted but hasn't been accepted yet in the mainline kernel, because it breaks the `FUSE` code which makes weird assumptions about the *misc* framework.

See http://linuxwell.com/2012/11/10/magical-container_of-macro/ for interesting implementation details about this macro.

Now, add code that copies (in a secure way) each character from the user space buffer to the UART device.

Once done, compile and load your module. Test that your `write` function works properly by using (example for UART2):

```
echo "test" > /dev/feserial-48024000
```

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

If it works, you can triumph and do a victory danse in front of the whole class!

Make sure that both UART devices work on the same way.

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.

Going further: ioctl operation

Do it only if you finish ahead of the crowd!

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 test applications (in source format) are already available in the `root/serial/` NFS shared directory. They assume that `SERIAL_RESET_COUNTER` is ioctl operation `0` and that `SERIAL_GET_COUNTER` is ioctl operation `1`.

Modify their source code according to the exact name of the device file you wish to use, and compile them on your host:

```
arm-linux-gnueabi-gcc -static -o serial-get-counter serial-get-counter.c
```

The new executables are then ready to run on your target.

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 Beaglebone
- Implement the read() operation of the serial port driver to put the process to sleep when no data are 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 misc driver lab*. Use the same kernel, environment and paths!

Register the handler

Declare an interrupt handler function stub. Then, in the module probe function, we need to register this handler, binding it to the right IRQ number.

Nowadays, Linux is using a virtual IRQ number that it derives from the hardware interrupt number. This virtual number is created through the `irqdomain` mechanism. The hardware IRQ number to use is found in the device tree.

First, add an `irq` field to your `feserial_dev` structure:

```
struct feserial_dev {
        struct miscdevice miscdev;
        void __iomem *regs;
        int irq;
};
```

Now, there is a very convenient shortcut to retrieve the hardware IRQ number from the device tree and get a virtual IRQ number.

```
dev->irq = irq_of_parse_and_map(pdev->dev.of_node, 0);
```

Then, pass the newly created virtual interrupt to `request_irq()` along with the interrupt handler to register your interrupt in the kernel.

Then, in the interrupt handler, just print a message and return `IRQ_HANDLED` (to tell the kernel that we have handled the interrupt).

You'll also need to enable receive interrupts. To do so, in the `probe()` function, set the `UART_IER_RDI` bit in the `UART_IER` register.

Compile and load your module. Send a character on the serial link (just type something in the corresponding `picocom` terminal, and look at the kernel logs: they are full of our message indicating that interrupts are occurring, even if we only sent one character! It shows you that interrupt handlers should do a little bit more when an interrupt occurs.

Enable and filter the interrupts

In fact, the hardware will replay the interrupt until you acknowledge it. Linux will only dispatch the interrupt event to the rightful handler, hoping that this handler will acknowledge it. What we experienced here is called an `interrupt flood`.

Now, in our interrupt handler, we want to acknowledge the interrupt. On the UART controllers that we drive, it's done simply by reading the contents of the `UART_RX` register, which holds the next character received. You can display the value you read to see that the driver will receive whatever character you sent.

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

Load and unload your driver multiple times, to make sure that there are no registration / deregistration issues.

Note: pay attention to where you place the call to `free_irq()`. Think about possible race conditions (typically because of resources not unregistered or freed in the right order, and the interference of user-space requests or hardware events happening at the same time). Don't hesitate to discuss the question with your instructor.

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 add a device-specific buffer to our `feserial_dev` structure.

Let's also add 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:

```
#define SERIAL_BUFSIZE 16

struct feserial_dev {
        void __iomem *regs;
        struct miscdevice miscdev;
        int irq;
        char serial_buf[SERIAL_BUFSIZE];
        int serial_buf_rd;
        int 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, add a wait queue to our `feserial_dev` structure, named for example `serial_wait`. In the `read()` function, keep things simple by directly using `wait_event_interruptible()` right from the start, to wait until `serial_buf_wr` is different from `serial_buf_rd`. [14].

Last but not least, 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/feserial-48024000` 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.

[14] A single test in the `wait_event_interruptible()` function is sufficient. If the condition is met, you don't go to sleep and read one character right away. Otherwise, when you wake up, you can proceed to the reading part.

Locking

Objective: practice with basic locking primitives

During this lab, you will:

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

Setup

Continue to work with the `feserial` driver.

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

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 debugging mechanisms and kernel crash analysis

Objective: Use kernel debugging mechanisms and analyze a kernel crash

In this lab, we will continue to work on the code of our serial driver.

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 does not have `CONFIG_DYNAMIC_DEBUG` set and your driver is not compiled with `DEBUG` defined., 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 `~/felabs/linux/modules/nfsroot/root/debugging/` directory.

Make sure your kernel is built with the following options:

- The `CONFIG_DEBUG_INFO` configuration option, (`Kernel Hacking` section) which makes it possible to see source code in the disassembled kernel.
- The `CONFIG_ARM_UNWIND` configuration option (`Kernel Hacking` section) disabled. 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.

Compile the `drvbroken` module provided in `nfsroot/root/debugging`.

On your board, load the `drvbroken.ko` 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 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:

```
cd ~/felabs/linux/src/linux/
arm-linux-gnueabi-objdump -S vmlinux > vmlinux.disasm
```

or, using `gdb-multiarch`[15]

```
sudo apt-get install gdb-multiarch
gdb-multiarch vmlinux
(gdb) set arch arm
(gdb) set gnutarget elf32-littlearm
(gdb) disassemble function_name
```

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.

[15] 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.

Going further: Git

Objective: Get familiar with git by contributing to the Linux kernel

After this lab, you will be able to:

- Explore the history of a Git repository
- Create a branch and use it to make improvements to the Linux kernel sources
- Make your first contribution to the official Linux kernel sources
- Rework and reorganize the commits done in your branch
- Work with a remote tree

Setup

Go to your kernel source tree in `~/felabs/linux/src/linux`

Exploring the history

With `git log`, look at the list of changes that have been made on the scheduler (in `kernel/sched/`).

With `git log`, look at the list of changes and their associated patches, that have been made on the ATMEL serial driver (`drivers/tty/serial/atmel_serial.c`) between the versions 3.0 and 3.1 of the kernel.

With `git diff`, look at the differences between `fs/jffs2/` (which contains the JFFS2 filesystem driver) in 3.0 and 3.1.

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

On the *cgit* interface of Linus Torvalds tree, available at https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/, search all commits that have been done by Free Electrons (hint: use the search engine by author).

Modify the Linux kernel sources

Find something to modify in the Linux kernel sources. Here are ideas:

- Choose an ARM defconfig file, apply it, run `make` and fix compile warnings
- Implement changes recommended in the Kernel Janitors page: http://kernelnewbies.org/KernelJanitors/Todo
- Run the `scripts/checkpatch.pl` command on a subdirectory of the Linux tree. You can do that with http://free-electrons.com/labs/run-checkpatch
- Remove deprecated features. For example, `IRQF_DISABLED` no longer does anything, but is still in use in many drivers. Find drivers that use this symbol, and fix them.
- Look for spelling mistakes in documentation, or classical mistakes like "the the", "a a"...

Before making changes, create a new branch and move into it.

Now, implement your changes, and commit them, following instructions in the slides for contributing to the Linux kernel.

Share your changes

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

Then, to send your patches, you will need to use your own SMTP server, either your company's if it is accessible from where you are, or the SMTP server available for a personal e-mail accounts (Google Mail for example, which has the advantage that your e-mail can be read from anywhere).

Configure git to tell it about your SMTP settings (user, password, port...).

Once this is done, send the patches to yourself using `git send-email`.

Check your changes

Before a final submission to the Linux kernel maintainers and community, you should run the below checks:

- Run `scripts/checkpatch.pl` on each of your patches. Fix the errors and warnings that you get, and commit them.
- Make sure that your modified code compiles with no warning, and if possible, that it also executes well.
- Make sure that the commit titles and messages are appropriate (see our guidelines in the slides)

If you made any change, use `git rebase --interactive master` to reorder, group, and edit your changes when needed.

Don't hesitate to ask your instructor for help. The instructor will also be happy to have a final look at your changes before you send them for real.

Send your patches to the community

Find who to send the patches to, and send them for real.

Don't be afraid to do this. The Linux kernel already includes changes performed during previous Free Electrons kernel sessions!

Unless you have done this before, you made your first contribution to the Linux kernel sources! We hope that our explanations and the power of git will incite you to make more contributions by yourself.

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/rt/linux-stable-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/
```

