Embedded Linux Development

Lab Book

Witekio
http://witekio.com

Based on training materials from Free Electrons
http://free-electrons.com

October 17, 2022

About this document

This document contains the practical exercices for this training session.

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 this course, 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 linux-for-schools-labs.tar.xz
sudo tar Jvxf linux-for-schools-labs.tar.xz
sudo chown -R <user>.<user> linux-for-schools-labs
```

Note that `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 `linux-for-schools-labs` 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, configuring the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.

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

Lab 1 : Linux rights

Playing around with linux file permissions

Objective: understand when to use `sudo` command.

Setup

Create an `hello_world` directory. In that directory create an `hello_world.c`, and implement a simple program which print ”Hello World!”. Simply compile it with:

```
$ gcc -o hello_world hello_world.c
```

Inspecting file permissions

Linux is a multi-user operating system, so files can be accessed by different users. 3 different rights can be given:

- read a file
- write a file
- execute a file

In order to manage who have the rights to access the files, Linux defines 3 entities:

- Users: there are defined in `/etc/passwd`
- Groups: there are defined in `/etc/group`
- Others

A file belongs to both a user and to a group. The user, the member of the group and other users can try to access the file. So the 3 different rights are defined for each of the 3 entities. It means that, if a file belong to you, you will have the rights defined for the user. If the file belong to a group and you are part of this group, you will have the rights defined for the group. If the file don’t belong to you and you are not part the group, you will have the rights defined for the others.

So who are you? You can know with which user you are logged in with the command `whoami`. Then to know which rights you have on the hello world files you can just use the `ls` command as follow:

```
$ ls -l
total 20
-rwxrwxr-x 1 formation formation 15960 oct.  12 11:33 hello_world
-rw-rw-r-- 1 formation formation    95 oct.  12 11:33 hello_world.c
```

The third string indicates which user is the owner of the file. The fourth string indicates which group is the owner of the file. The first string indicates which rights have formation user, formation group and the others. Let’s have a look to the `hello_world.c` file. The string began with a `-`, it indicates that it is a regular file. Then it is followed by `rw-`. This is the right for the user formation who have the right to read and write the file, but not to execute it. Then it is followed by `rw-`. It is the right for the group who also have the right to read and write the file, but not to execute it. Finally there is `r--`. It is the right for the others who have only the right to read the file.

Modifying file permissions

You can modify the file permissions on your own files. The command `chmod` is done for it. Access rights are often given using its octal form:

- read = 4
- write = 2
- execute = 1

So if you want to give read, write and execute permissions, it is `4 + 2 + 1 = 7`. You must set the rights for the 3 entities, so if you want to give read, write and execution permissions to everybody you will use `777` argument with `chmod` command.

```
$ chmod 777 <my file>
```

Let 's try it removing the right to execute the file `hello_world`. Are you still able to run the `hello_world` program?

Asking a user to do something for you

On Linux systems there is a special user: `root`. This is a privileged user, so `root` will have a lot of rights on your system. A lot of files of your system belong to `root`. Just have a look to `/etc/passwd` for example. Normally it belongs to `root`, and hopefully only `root` can modify it.

Generally not having the rights to modify this kind of file is a good idea to prevent breaking the system. But sometimes you can need to do some actions that only `root` can do, like installing packages or writing on block device. The `sudo` command is made to ask another user to do something for you. By default it asks to `root` to do it. Who have the rights to ask `root` to do something is defined by a security policy generally defined by the `/etc/sudoers` file. On Ubuntu, ”Administrator” user will have that right.

Let's try to use that command. First remove `hello_world` program:

```
$ rm hello_world
```

Then ask `root` to recompile the hello world program:

```
$ sudo gcc -o hello_world hello_world.c
```

Inspect the file permissions on the binary you just created. Are you able to execute it? To remove it?

Ok, now, let's create a build directory to put that binary:

```
$ sudo mkdir build
$ sudo gcc -o build/hello_world hello_world.c
```

Are you still able to remove that file and directory? In latter labs, you would have to use system baker to create a Linux system from scratch. If you use this command with `root` it will create a lot of files and directories that your regular user won't be able to remove. So don't use `sudo` command if you don't need it, and using a system baker to build a Linux system don't need it!

Changing ownership

If, by mistake, you use `sudo` when you create a file or directory, it is still possible to change the ownership of that file or directory. It is done with the command `chown`. This command can act recursively under subdirectories thanks to the `-R` option and the arguments taken by this command are the user and the group separated by a colon and the path of the file or directory. Let's try to get back the build directory created previously:

```
$ sudo chown -R formation:formation build
```

Inspect the file permissions to check, that now, you are the owner of that directory and that file.

Conclusion

Using `root` user give you a lot of rights, even the one to break your system. A lot of task don't need such rights and can be achieved with regular user rights. So don't use `root` user and `sudo` command if you don't really need it. And, if you need it, be careful with what you are doing.

Lab 2 : First Buildroot build

First dive into the Buildroot project and its build mechanism

Objective: Discover how a build system is used and how it works, with the example of the Buildroot build system. Build a Linux system with libraries.

Setup

Create the `$HOME/embedded-linux-labs/buildroot` directory and go into it.

Get Buildroot and explore the source code

The official Buildroot website is available at `http://buildroot.org/`. Download the latest LTS 2022.02.6 version which we have tested for this lab. Uncompress the tarball and go inside the Buildroot source directory.

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

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

Configure Buildroot

In our case, we would like to:

- Generate an embedded Linux system for the i.MX6q SabreLite board
- Integrate *Busybox*, *alsa-utils* and *vorbis-tools* in our embedded Linux system;
- Integrate the target filesystem into a tarball

First apply the default configuration for the i.MX6q SabreLite board.

```
make nitrogen6x_defconfig
```

To run the configuration utility of Buildroot, simply run:

```
make menuconfig
```

Set the following options. Don't hesitate to press the `Help` button whenever you need more details about a given option:

- `Toolchain`
  - `C library`
    - Select `glibc`
  - Select `Build cross gdb for the host`
- `Target packages`
  - Keep `BusyBox` (default version) and keep the Busybox configuration proposed by Buildroot;
  - `Audio and video applications`
    - Select `vorbis-tools`
  - `Debugging, profiling and benchmark`
    - Select `gdb`
    - Select `gdbserver`
  - `Networking applications`
    - Select `dropbear`
      - Select `client programs`
      - Select `optimize for size`
  - `Games`
    - Select `sl`
- `Filesystem images`
  - Select `tar the root filesystem`
  - `Compression method`
    - Select `bzip2`

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

Generate the embedded Linux system

Just run:

```
make
```

Buildroot will first create a small environment with the external toolchain, then download, extract, configure, compile and install each component of the embedded system.

All the compilation has taken place in the `output/` subdirectory. Let's explore its contents:

- `build`, is the directory in which each component built by Buildroot is extracted, and where the build actually takes place
- `host`, is the directory where Buildroot installs some components for the host. As Buildroot doesn't want to depend on too many things installed in the developer machines, it installs some tools needed to compile the packages for the target. In our case it installed *pkg-config* (since the version of the host may be ancient) and tools to generate the root filesystem image (*genext2fs*, *makedevs*, *fakeroot*).
- `images`, which contains the final images produced by Buildroot. In our case it's just a tarball of the filesystem, called `rootfs.tar`, but depending on the Buildroot configuration, there could also be a kernel image or a bootloader image.
- `staging`, which contains the "build" space of the target system. All the target libraries, with headers and documentation. It also contains the system headers and the C library, which in our case have been copied from the cross-compiling toolchain.
- `target`, is the target root filesystem. All applications and libraries, usually stripped, are installed in this directory. However, it cannot be used directly as the root filesystem, as all the device files are missing: it is not possible to create them without being root, and Buildroot has a policy of not running anything as root.

Lab 3 : Create an SD card

Objective: understand how the Linux system is copied on a SD card.

Block devices

On Linux system the `/dev` directory is the one which contains the ”device” files. These files represent the hardware. So there are some files there to give an interface to your hard disk drives, your USB keys, or your SD cards read by an SD card reader. All these devices made to store some data are generally called `/dev/sdX`, where `X` should be replaced by a lower-case letter. This letter indicates the order in which the devices have been found following the alphabetic order.

So let have a look to the block device already present on our system:

```
$ ll /dev/sd*
brw-rw---- 1 root disk 8, 0 oct.  13 13:53 /dev/sda
brw-rw---- 1 root disk 8, 1 oct.  13 13:53 /dev/sda1
brw-rw---- 1 root disk 8, 2 oct.  13 13:53 /dev/sda2
brw-rw---- 1 root disk 8, 3 oct.  13 13:53 /dev/sda3
```

We can see that there is one disk: `/dev/sda`. What about the other files? Each of them represent a partition of the disk `/dev/sda`.

Note: these files belong to the root user and the disk group. If your user is not part of that group, you will need to use `sudo` command to read and write these files.

Now, plug the SD card reader with your SD card, and have a look to the block device. Any changes? You can also have a look to the messages from the Linux kernel:

```
$ sudo dmesg
...
[ 3024.806284] sd 33:0:0:1: [sdc] 7744512 512-byte logical blocks:
(3.97 GB/3.69 GiB)
...
[ 3024.851013]  sdc: sdc1
[ 3024.860086] sd 33:0:0:1: [sdc] Attached SCSI removable disk
```

It seems that a disk `/dev/sdc` has been found and that disk contains one partitions.

Warning: your setup may me a little different, so we will refer to your SD card device as `/dev/sdX`. Just replace in the next sections the `X` letter by the appropriate value.

Mount a partition

To access the filesystem on a partition, you can use `mount` command. With no argument it shows the lists of mounted filesystem. Just try it. Where is mounted the filesystem on the partition of your SD card? Have a look to these files.

To unmount a filesystem you can use the command `umount` with the path to the mount point or the device file containing the filesystem. Try to unmount the filesystem which is on the first partition of your SD card.

In order to mount a filesystem you can use `mount` command. The first argument is the device file, the second argument is the mount point:

```
$ sudo mount /dev/sdX1 /mnt
```

Check that the filesystem on the first partition of the SD card is in the list of mounted filesystem. Have a look to these files.

Table of partitions

The first 512 octets of a disk is called the MBR which stands for Master Boot Record. It is in that block that you can found the partition table of the disk. Let's have a look. First unmount the first partition. Then display the content of that first block with the following command:

```
$ sudo hexdump -C /dev/sdX -n 512
```

Partition entry is 16 bytes long. The first partition entry is at the offset `0x01BE`. So the second entry is at the offset `0x01CE`. Any difference?

It it possible to copy the values contained into the MBR. We can do it with the `dd` command. Let's do it. We will copy from the device file which represent our SD card to a regular file called `MRB.bak` 1 block of 512 bytes:

```
$ sudo dd if=/dev/sdX of=MBR.bak count=1 bs=512
```

Have a look to that file using `hexdump`. Now we want to erase the MBR of the SD card. We will copy 1 block of 512 bytes of zeros to the disk.

```
$ sudo dd if=/dev/zero of=/dev/sdX count=1 bs=512
```

Have a look to the MBR of the SD card using`hexdump`. Where is the `/dev/sdX1` file?

It is possible to copy back the content of the MBR from the backup we did. But we can also create a new partition table using the tool `parted`. Let's have a look to this second command. First of all we will recreate the MBR with an empty table partition:

```
$ sudo parted /dev/sdX mklabel msdos
```

Have a look to the MBR. There is no partition entry, so the `/dev/sdX1` is still missing. So we will create a first partition which will take the full size of the disk:

```
$ sudo parted -a optimal /dev/sdX mkpart primary 0% 100%
```

Is the `/dev/sdX1` back? What about the entry of the first partition into the MBR?

Conclusion

`/dev/sdX` represent a disk, `/dev/sdX1` the first partition of that disk, `/dev/sdX2` the second partition of that disk... Generally a standard user won't have the permissions to read or write these files, so you will need to use the `sudo` command.

System bakers can generate an image of the whole disk, an image of the partitions, a `tar` archive... In the first case, you can just copy the data on the SD card with the `dd` command. In the second case, you will have to create a partition table thanks to `parted` or `fdisk` command, then copy the content of the partition with the `dd` command. In the case of an archive, after having created the partition you would have to format it with `mkfs` command, mount the partition and extract the `tar` archive into the partition. So there is multiple ways to create a SD card containing your new Linux system, it depends on the output of the system baker.

Lab 4 : First target run

Boot using a SD Card, then using TFTP/NFS

Objective: Learn how to flash and boot your Sabre Lite board.

During this lab, you will:

- Flash a full system image on an SD Card.
- Launch Embedded Linux on your Sabre Lite.
- Get the kernel image using TFTP.
- Mount a remote root filesystem using NFS.

First contact with your Sabre Lite

Lets program a system image to your board SD Card.

Plug the SD Card to prepare on your PC and check the associated device.

Example:

```
$ dmesg
[ 1824.281873] sdb: sdb1
[ 1824.309313] sd 3:0:0:0: [sdb] Assuming drive cache: write through
[ 1824.309384] sd 3:0:0:0: [sdb] Attached SCSI removable disk
```

In the case above, the SD Card has been associated to `/dev/sdb` and has one partition exposed as `/dev/sdb1`.

From now on, we will refer to your SD Card device as `/dev/sdx`. Just replace `x` with the correct device letter in the following commands.

Use the `mount` command to check if the SD Card has been automatically mounted:

```
$ mount
...
/dev/sdb1 on /media/460A-AB1E type vfat
(rw,nosuid,nodev,uhelper=udisks,uid=1000,gid=1000,shortname=mixe
d,dmask=0077,utf8=1,showexec,flush)
```

If any partition of the SD Card appears to be mounted, unmount it as follows:

```
$ sudo umount /dev/sdx*
```

Create a partition on the SD card to host the root filesystem, then write the generated image to it:

```
$ sudo parted /dev/sdx mklabel msdos
$ sudo parted -a optimal /dev/sdx mkpart primary 0% 100%
$ cat output/images/rootfs.ext2 | sudo dd of=/dev/sdx1 bs=1M
```

On the development PC, run `picocom`:

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

Use `/dev/ttyUSB0` if you are using a USB to serial converter, `/dev/ttyS0` if using a standard serial port on your PC. To exit `picocom`, press `[Ctrl][a]` followed by `[Ctrl][x]`.

If the system does not permit you access to the tty device, check that you are in the group owning it.

Insert the SD Card into the board's SD slot and power it on. The kernel should boot and then mount the root filesystem.

If it is not the case, enter the U-Boot prompt and revert its environment with:

```
U-Boot > env default -a
U-Boot > setenv board nitrogen6x
U-Boot > saveenv
U-Boot > reset
```

You should end up with a command-line prompt in the `picocom` window, enter `root` as the login to enter the shell.

Booting with TFTP and NFS

U-Boot can also load the kernel from a TFTP server instead of reading it out of the SDCard. This is very useful during development phase to fasten up the process of deploying a newly built kernel image.

Install a TFTP server on your development workstation:

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

We can also have the root filesystem mounted from a remote location on the network, which is useful when stored on the development PC to make quick changes to it.

Install a NFS server on your development workstation:

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

Edit file `/etc/exports` on the host system to add the following line:

```
/srv/nfs/rootfs *(rw,sync,insecure,no_root_squash,no_subtree_check)
```

Create the link from the NFS server to the root filesystem and copy the kernel and device tree images to the TFTP directory.

```
$ sudo cp output/images/zImage /srv/tftp/
$ sudo cp output/images/imx6q-sabrelite.dtb /srv/tftp/
$ sudo mkdir -p /srv/nfs/rootfs
$ sudo tar xvf output/images/rootfs.tar.bz2 -C /srv/nfs/rootfs
```

Restart the NFS server:

```
$ systemctl restart nfs-kernel-server.service
```

Reset the board and interrupt the boot process by pressing a key on U-Boot's countdown. Modify the following variables to have the board boot from ethernet.

```
U-Boot > setenv ethaddr <mac_addr_of_the_board>
U-Boot > setenv serverip <ip_addr_of_the_dev_pc>
U-Boot > setenv netargs 'setenv bootargs
  console=${console},${baudrate} root=/dev/nfs ip=dhcp
  nfsroot=${serverip}:${nfsroot},v3,tcp'
U-Boot > setenv nfsroot '/srv/nfs/rootfs/'
U-Boot > setenv netboot 'dhcp ${loadaddr} zImage; tftp ${fdt_addr}
  imx6q-sabrelite.dtb; bootz ${loadaddr} - ${fdt_addr}'
U-Boot > setenv bootcmd 'run netargs; run netboot'
U-Boot > saveenv
```

If you want to boot using a static IP address for the board, change the following variables:

```
U-Boot > setenv ipaddr <ip_addr_of_the_board>
U-Boot > setenv netargs 'setenv bootargs
  console=${console},${baudrate} root=/dev/nfs ip=${ipaddr}
  nfsroot=${serverip}:${nfsroot},v3,tcp'
U-Boot > setenv netboot 'tftp ${loadaddr} zImage; tftp ${fdt_addr}
  imx6q-sabrelite.dtb; bootz ${loadaddr} - ${fdt_addr}'
```

Boot the kernel from TFTP by typing:

```
U-Boot > boot
```

Lab 5 : Buildroot configuration

Configuring parts of the system through Buildroot

During this lab, you will:

- Modify basic Buildroot configuration using the configuration utility.
- Use Buildroot overlays.
- Use Buildroot custom scripts.

Buildroot basic customization

Let's do some basic modifications to Buildroot image, so:

- The hostname of the platform is set to something more descriptive than *Buildroot*.
- The network interface is automatically configured using DHCP.
- A root password is set, allowing to log through SSH.
- Remove an unwanted package.

First, start the Buildroot configuration utility:

```
cd buildroot/
make menuconfig
```

In `System configuration`, modify the following elements:

- Set `System hostname` to any valid hostname you like.
- Set `Network interface to configure through DHCP` to the name of our interface: `eth0`.
- Set `Root password` to any password. You may want to use a simple password during this training.

Now we want to remove the useless `sl` packages installed previously. In the `Target packages` menu, then `Games`, unselect `sl`.

You then need to rebuild the system. However, you may notice `sl` is not removed from the target rootfs with a simple build.

```
make
find output/target -name sl
```

This is because we're hitting one of the current limitations of Buildroot. To keep its design simple and efficient, Buildroot currently doesn't support removing software even if it has been removed from the configuration. The clean way is to run `make clean` and then run `make` again. However, you may want to skip this step during this lab to avoid the long compilation time.

Buildroot advanced customization

Up to this point we have been configuring Buildroot using the configuration menu. However, the configuration options are not infinite so two more mechanism are available:

- The ovelay directory.
- The custom scripts run before and after filesystem creations.

Using Buildroot overlay directory

Buildroot overlay directory allows you to use a set of files copied on your root filesystem once the compilation is done. Overlay directory has to follow the same architecture as the target rootfs directory. So a file named `<overlay>/foo/bar` will end in `/foo/bar` in the final image.

Here we would like to set a static IP address. This can be done using the `/etc/network/interfaces` file, using a syntax like the following example.

```
auto eth0
iface eth0 inet static
        address 192.0.2.7
        netmask 255.255.255.0
        gateway 192.0.2.254
```

To use a static IP:

- Create an overlay directory. Suggested name: `board/company_name/board_name/rootfs_overlay/`.
- In the configuration utily, set the overlay directory in `System configuration`, `Root filesystem overlay directories`.
- Do not forget to disable DHCP option activated in previous lab.
- Copy existing `output/target/etc/network/interfaces` in your overlay and modify its content to set a static IP. Do not remove the existing section for `lo` interface.

Once you have recompiled buildroot and redeployed the generated images, you can check your IP using the `ip address` command.

Using Buildroot custom scripts

Buildroot custom script allows you to run commands just before or just after the final image is created. This allows you add dynamic content on the rootfs or to support additional image format.

For now, all we want is to create a custom file named `/root/build_date` containing the result of the `date` command. This can be done using a simple script (suggested filename: `board/company_name/board_name/post-build.sh`) containing:

```
#!/bin/sh
date > $TARGET_DIR/root/build_date
```

Buildroot can be tell to use the script in the configuration utility, section `System configuration`: `Custom script to run before creating filesystem images`. Take care, there may already be a script defined. In such a case you will need to modify the existing script or to make sure to call it from your own.

You can then check the content of the file using `cat /root/build_date`.

Lab 6 : Add a custom application

Add a new recipe to support a required custom application

During this lab, you will:

- Add a custom application in Buildroot packages.
- Integrate this application into the build.

Setup and organization

In this lab we will add a buildroot config and a Makefile handling the `nInvaders` application. Before starting the recipe itself, find the `package` directory and add a subdirectory for your application.

First hands on nInvaders

The nInvaders application is a terminal based game following the space invaders family. In order to deal with the text based user interface, nInvaders uses the ncurses library.

First try to find the project homepage, download the sources and have a first look: license, Makefile, requirements…

Write the Config.in file

Create a configuration file named `Config.in`

In this file add the config definition: package name, description and dependencies. All these informations will be used by the configuration tool.

Add a reference to your new `Config.in` file in the existing `package/Config.in`.

You may base your `Config.in` file on the ones already present for existing packages or you may read `docs/manual/adding-packages-directory.txt`.

Write the .mk file

Now create a `ninvaders.mk` file: this will describe how buildroot can get the packages sources and build it.

In this file, define the following elements:

- Package version.
- Download site.
- Package file name.
- License.
- Build and install commands.

You may get help from `docs/manual/adding-packages-generic.txt` or by reading makefiles for existing packages.

Remember all variable need to be prefixed by the package name: `NINVADERS_`.

Update the rootfs and test

You can check the whole packaging process is working fine by explicitly running the build task on the `nInvaders` package:

```
make ninvaders
```

To include the program in the image, run `make menuconfig` and select nInvaders.

You can confirm the `nInvaders` program is present by running:

```
find output/target -iname ninvaders
```

You will now need to deploy your rootfs again, on the SD card or on the corresponding NFS directory.

Access the board command line through SSH. You should be able to launch the `nInvaders` program. It's time to play!

Lab 7 : Kernel cross-compiling

Cross-compile a kernel for a target board

Objective: Learn how to cross-compile a kernel for Sabre Lite.

During this lab, you will:

- Retrieve kernel sources for the Sabre Lite board.
- Set up a cross-compiling environment for the kernel.
- Configure the kernel.
- Cross compile the kernel.

Getting the sources

We need kernel sources with complete support for Sabre Lite. Unfortunately they are not completely mainlined, so we need to clone them from the vendor in charge of their maintainance.

In a directory created for this lab, run:

```
$ git clone https://github.com/boundarydevices/linux.git
$ cd linux
$ git checkout -b training origin/boundary-imx_5.10.x_2.0.0
```

Cross-compiling environment setup

To manually cross-compile the kernel, we will use the the cross-compiling toolchain provided by your system baker.

Add the cross-compiling toolchain path to your `PATH` environment variable.

Don't forget to export the correct `ARCH` and `CROSS_COMPILE` variables.

Linux kernel configuration

Pick up the default Linux configuration for the Sabre Lite board and apply it.

Don't hesitate to visualize the settings by running `make menuconfig` afterwards.

Cross compiling

Try to compile the kernel and the correct device tree for Sabre Lite (replace X by the number of CPU cores on your PC):

```
$ make zImage -jX
$ make modules -jX
$ make imx6q-sabrelite.dtb
```

Once the builds complete, try to locate the created images.

Booting your kernel

Time to boot your newly built kernel. Copy the kernel image and the device tree blob under the data folder of your TFTP server, then install the modules in the NFS rootfs. Finally reboot your platform and make sure it downloads the new kernel over TFTP.

If you can reach a command line shell, congratulations !

Lab 8 : Kernel source code

Objective: Get familiar with the kernel source code

After this lab, you will be able to:

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

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. [1]
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()` function is declared, implemented and even used.

[1]Look for files in `logo` in their name. It's an opportunity to practise with the `find` command.

Lab 9 : Writing modules

Create a simple kernel module

Objective: learn how to write and compile a basic module.

During this lab, you will:

- Write a simple kernel module with parameters.
- Access kernel internals from your module.
- Setup the environment to compile it.
- Add the sources of your module to the kernel source tree.
- Build a kernel patch with your modifications.

Lab implementation

You know that code from Linux kernel modules is treated just as any kernel code, and executed without any particular control, remember that any mistake in module code can corrupt or crash the running kernel.

This is particularly true with modules under development. If you used your PC workstation to test your new modules, you could face several critical failures, and waste a significant amount of time rebooting your system and restoring your development environment.

When writing modules for a target platform, it is better to directly test the module on the platform. This way if the module crashes the kernel, just reset the board to recover a running target system.

Writing a module

Create an `src` directory. When the compilation is done, you will have to deploy the `.ko` file into your rootfs, simply copying it into your NFS directory, or transferring it through SSH with `scp` command.

Create a `hello-version.c` file implementing a module which displays the following message when loaded:

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

and displays a goodbye message when unloaded.

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

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.

Create a `Makefile` that contains the following code to compile your module:

```
ifneq ($(KERNELRELEASE),)
obj-m := hello-version.o
else
KDIR := <path_to_your_kernel_sources>
PWD := $(shell pwd)
all:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
endif
```

Note: The `Makefile` syntax does not like spaces for indentation. Make sure you use only tabs when indenting.

Building your module

Launch your module compilation and make sure to fix any error or warning. Do not forget to set the `ARCH` and `CROSS_COMPILE` environment variables prior to launch the build.

Testing your module

Load your new module from the target board. Check that it works as expected.

If it does not work, unload it, modify its code, compile and load it again as many times as needed.

Run the appropriate command to check that your module is on the list of loaded modules. Now, try to get the list of loaded modules only using 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 `ktime_get_seconds()` function to achieve this.

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

Following Linux coding standards

As many projects, the kernel has its own coding standards. If you want to have your code one day merged in the mainline sources, it should adhere to these standards.

Fortunately, the Linux kernel community provides a utility to find coding standards violations, `scripts/checkpatch.pl` in the kernel sources.

To find the available options:

```
$ <path_to_your_kernel>/scripts/checkpatch.pl -h
```

Then to check for violations in `hello-version.c`:

```
$ <path_to_your_kernel>/scripts/checkpatch.pl --file --no-tree \
  hello-version.c
```

Fix any violation found in your code. If there are many indentation related errors, make sure you use a properly configured source code editor, according to the kernel coding style rules in `Documentation/CodingStyle`.

Adding your module to the kernel sources

As we are going to make changes to the kernel sources, first create a specific branch for them.

First make sure you are on the `training` branch you created earlier:

```
$ git status
```

Then create the specific branch:

```
$ git checkout -b hello
```

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

Check that the configuration interface shows your addition and lets you configure it as a module. Run the `make` command and make sure that the code of your module gets compiled.

Then commit your changes in the current branch (with an appropriate commit message):

```
$ git add -A
$ git commit -as
```

- `git add -A` adds modifications 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 modifications, is to run `git status` and explicitly run `git add` on each files containing modifications you want to add to the next commit.
- `git commit -a` creates a commit with all the validated modifications (with `git add`) 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

To be able to share your modifications with others, create a patch containing them.

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 derived from:

```
$ git format-patch training
```

Have a look at the generated file. You can see that its name reuse 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.