Embedded Linux networking Training

Espressobin variant

Practical Labs

https://bootlin.com

July 03, 2025

About this document

Updates to this document can be found on `https://bootlin.com/doc/training/networking`.

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

More details about our training sessions can be found on `https://bootlin.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 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 https://bootlin.com/doc/training/networking/networking-labs.tar.xz
$ tar xvf networking-labs.tar.xz
```

Lab data are now available in an `networking-labs` directory in your home directory. This directory contains directories and files used in the various practical labs. It will also be used as working space, in particular to keep generated files separate when needed.

Update your distribution

To avoid any issue installing packages during the practical labs, you should apply the latest updates to the packages in your distro:

```
$ sudo apt update
$ sudo apt dist-upgrade
```

You are now ready to start the real practical labs!

Install extra packages

Feel free to install other packages you may need for your development environment. In particular, we recommend to install your favorite text editor and configure it to your taste. The favorite text editors of embedded Linux developers are of course *Vim* and *Emacs*, but there are also plenty of other possibilities, such as Visual Studio Code[1], *GEdit*, *Qt Creator*, *CodeBlocks*, *Geany*, etc.

It is worth mentioning that by default, Ubuntu comes with a very limited version of the `vi` editor. So if you would like to use `vi`, we recommend to use the more featureful version by installing the `vim` package.

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.

[1]This tool from Microsoft is Open Source! To try it on Ubuntu: `sudo snap install code --classic`

- 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: `$ sudo chown -R myuser.myuser linux/`

Setup the build environment

Objective: Configure our testing setup for the labs

Goals

- Have an overview of the build setup
- Run our first buildroot build
- Setup the Host machine

Install the required packages

You need some packages installed on your host machine :

```
$ sudo apt install build-essential git tcpdump wireshark iperf3 python3-scapy ethtool \
  clang linux-tools-common libbpf-dev pahole
```

Getting started with Buildroot

For this training session, we will be running a pre-configured Linux OS image that we will generate usnig Buildroot version 2025.02

The Espressobin V7 is already well supported in Buildroot, the image we are using contains a few extra setup options that will make the next labs easier to run :

- The Linux Kernel version is v6.12.y, the latest LTS release
- Some networking related packages are pre-installed, namely :
  – ethtool, for low-level network interface configuration (see `man 8 ethtool`)
  – iperf3, for traffic generation (see `man 1 iperf3`)
  – iproute2, replacing the busybox-based implementation, for network configuration (see `man 8 ip`)
  – tcpdump, for traffic analysis (see `man 8 tcpdump`)
  – Custom packages for the various labs.

Our configuration also includes an overlay directory, which will allows us to very easily install custom-made files into our rootfilesystem.

Understanding the Lab materials

In this training, we will be running some commands both on the Espressobin and the Host machine.

In order to quickly see where a given command should run, we have color-coded the instructions.

Target commands

Commands on a Yellow background are to be run on the Espressobin, also referred to as **Target** :

```
$ echo "Hello world, I am an Espressobin"
```

For simplicity, the expected output of commands running on the Espressbin also appear on a yellow background :

```
Hello world, I am an Espressobin
```

Target commands

Commands that needs to run on your **Host** machine are on a Green background :

```
$ echo "Hello world, I am the host machine"
```

For simplicity, the expected output of commands running on the Espressbin also appear on a yellow background :

```
Hello world, I am the host machine
```

Setup the host machine for the neworking labs

You can use a built-in Ethernet port n your host machine, provided that it is not in use, and that it is capable of 1Gbps speed. This can be checked with :

```
$ethtool <host_iface>
       settings for enp0s20f0u1u1:
       Supported ports: [ TP MII ]
       Supported link modes: 10baseT/Half 10baseT/Full
                             100baseT/Half 100baseT/Full
                             1000baseT/Half 1000baseT/Full
       Supported pause frame use: No
       Supports auto-negotiation: Yes
       Supported FEC modes: Not reported
       ...
```

If you do not have such an interface, you can use the provided USB to Ethernet adapter. It supports 1Gbps, but its driver doesn't report that.

We will be doing some manual re-configuration of the host interface. Regardless if you chose to use the built-in interface, you need to make sure that NetworkManager will not try to re-configure your interface, thus overriding your configuration. You can achieve that temporarily by running :

```
$ nmcli device set <iface> managed no
```

This only temporary, as the interface will become managed again when your host machine reboots.

Building our image

Let's now build our buildroot image :

```
$ cd /home/$USER/networking-labs/buildroot
$ make globalscale_espressobin_networking_defconfig
$ make
```

This should take a while :) The buildroot image provided is also hosted on our github, you can take a look here for more details.

Preparing the Espressobin

The Espressobin is powered by a 12V DC external PSU. Make sure that your PSU is rated for 2A, and has a center-positive Barrel Jack.

In addition, to access the debug serial console, you need to use a micro-USB cable connected to the micro-USB port, near the Barrel Jack.

Once your micro-USB cable is connected, a `/dev/ttyUSB0` device will apear on your PC. You can see this device appear by looking at the output of `dmesg` on your workstation.

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

```
sudo apt 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
```

Important: for the group change to be effective, you have to *completely reboot* the system [2]. A workaround is to run `newgrp dialout`, but it is not global. You have to run it in each terminal.

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.

Flashing the SDCard

You will now need to flash a sdcard with the `output/images/sdcard.img` file. Plug your sdcard on your computer and check on which `/dev/sdX` it has been mounted (you can use the `dmesg` command to check that). For instance, if the sdcard has been mounted on `/dev/sde`, use the following command:

```
$ sudo dd if=output/images/sdcard.img of=/dev/sde
$ sync
```

NOTE: Double-check that you are targeting the correct device before executing the dd command!

Once flashed, insert the sdcard into the Espressobin.

[2] As explained on https://askubuntu.com/questions/1045993/after-adding-a-group-logoutlogin-is-not-enough-in-18-04/.

Interacting with the Networking Stack

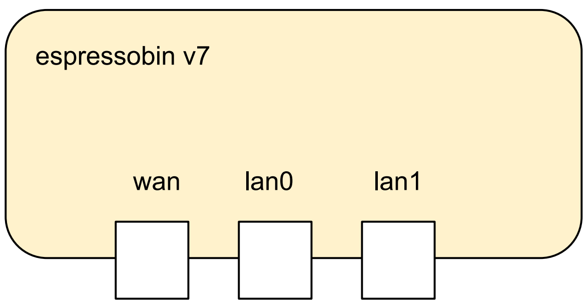
Objective: learn how to manage Networking interfaces and create new ones

Goals

- Get familiar with the Espressobin's network interfaces
- Use `iproute2` for link configuration, vlan setup and bridiging
- Use the different VLAN configuration methods
- Use network namespaces for interface isolation

Getting familiar with the Espressobin

The Espressobin has 3 front-facing ports labelled "wan", "lan0" and "lan1" :



List the network interfaces by running :

```
$ ip link show
```

Do you notice anything strange ? Besides the `lo` interface for loopback, we see 4 interfaces :

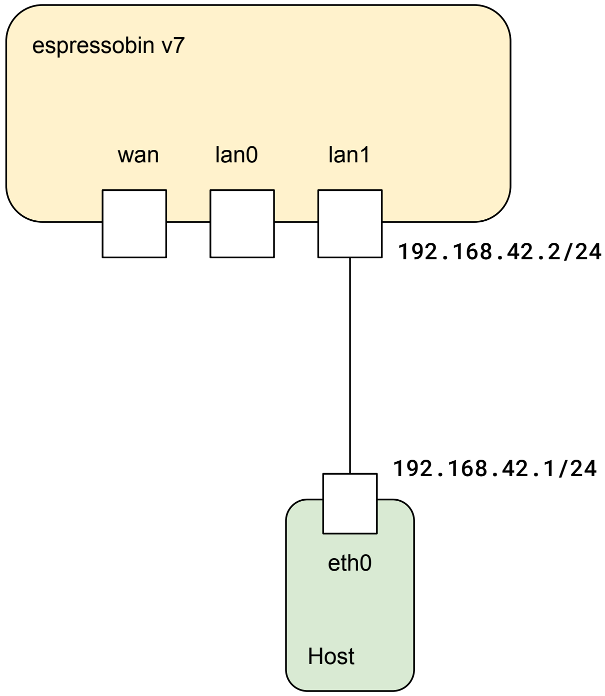
- `eth0`
- `lan0@eth0`
- `lan1@eth0`
- `wan@eth0`.

This is because the Espressobin uses a dedicated **DSA switch** to drive its ports, and `eth0` is used as conduit between the **SoC's Ethernet Interface** and one of the switch's ports.

In this lab, we will ignore `eth0` and only focus on the other 3 interfaces. Although they appear as `lan0@eth0`, you must use the shorter `lan0` interface name in your commands.

Sending our first packet

Let's send out first ping between our Host machine and the Espressobin. This is the setup we'll achieve :



The first step we need to take is to administratively bring the `lan1` interface **up** :

```
$ ip link set lan1 up
```

You can check that an interface is "admin UP" by running :

```
$ ip link show lan1
3: lan1@eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> ...
```

The "UP" keywork appears as the last element betweek the chevrons following the interface name, this is the admin link state. When the interface is "admin down", no keywork will appear in that spot.

The next step is to check that we have a working connection with the Host. Plug the Ethernet cable between the `lan1` port and your Host machine. Make sure that your Host's interface is UP as well. Let's check the link status one more time :

```
$ ip link show lan1
lan1@eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP ...
```

The `LOWER_UP` indication tells you that there's a **link-partner** detected, using **PHY Layer** information when available. The `state UP` keywords indicates that the whole link between this interface and the link partner is up and running, ready to transmit traffic.

To get information about the PHY layer attributes, you can use **ethtool** :

```
$ ethtool lan1
```

```
Settings for lan1:
	Supported ports: [ TP MII ]
	Supported link modes:   10baseT/Half 10baseT/Full
	                        100baseT/Half 100baseT/Full
	                        1000baseT/Full
	Supported pause frame use: Symmetric
	Supports auto-negotiation: Yes
	Supported FEC modes: Not reported
	Advertised link modes:  10baseT/Half 10baseT/Full
	                        100baseT/Half 100baseT/Full
	                        1000baseT/Full
	Advertised pause frame use: Symmetric
	Advertised auto-negotiation: Yes
	Advertised FEC modes: Not reported
	Link partner advertised link modes:  10baseT/Half 10baseT/Full
	                                     100baseT/Half 100baseT/Full
	                                     1000baseT/Full
	Link partner advertised pause frame use: Symmetric Receive-only
	Link partner advertised auto-negotiation: Yes
	Link partner advertised FEC modes: Not reported
	Speed: 1000Mb/s
	Duplex: Full
	Auto-negotiation: on
	Port: Twisted Pair
	PHYAD: 17
	Transceiver: external
	MDI-X: Unknown
	Supports Wake-on: d
	Wake-on: d
	Link detected: yes
```

We can get from the above that :

- The link is **up** : `Link detected: yes`
- The link speed was established at 1Gbps : `Speed: 1000Mb/s`
- The Link-partner (Host machine) supports 10, 100 and 1000Mbps links

To send our first ping, we now need to assign IPv4 addresses to both `lan1` as well as our Host's interface. Let's use the `192.168.42.0/24` subnet :

```
$ ip address add 192.168.42.2/24 dev lan1
```

You can show the currently assigned IP addresses to each interface by running :

```
$ ip address
```

Run a similar command on your host machine to assign it the `192.168.42.1/24` address.

You can now send your first **ping**, running the following command on your Host machine :

```
$ ping 192.168.42.2
```

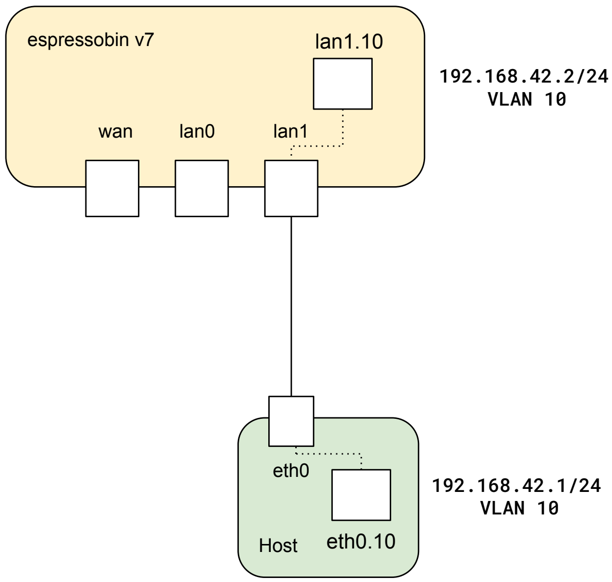
Create a VLAN

Now that we know that the link between our Host machine and the Espressobin works, let's setup a VLAN link.

First, remove any prior IPv4 configuration we've done on `lan1` :

```
$ ip address flush lan1
```

The setup we want to achieve is the following :



We'll use a dedicated netdevice for the VLAN we want to create. We will use VLAN id 10, which is arbitrary here. We assign an IP address to the vlan interface, and set it as admin-up :

```
$ ip link add link lan1 name lan1.10 type vlan id 10
$ ip address add 192.168.42.2/24 dev lan1.10
$ ip link set lan1.10 up
```

Configure your Host machine in the same way, with the same VLAN id, but adjusting the IP address.

You should now be able to ping your Espressobin from your Host. Leave the `ping` command running, we'll need some traffic to go through for this next part.

Let's go a bit further and capture some traffic, to see what the encapsulation looks like. For now, we'll only use `tcpdump` as our capturing software.

```
$ tcpdump -n -e -i lan1.10
```

You should see the ICMP traffic generated by the `ping` command running on your host :

```
[...] ethertype IPv4 (0x0800), length 98: 192.168.42.1 > 192.168.42.2: ICMP echo request, ...
[...] ethertype IPv4 (0x0800), length 98: 192.168.42.2 > 192.168.42.1: ICMP echo reply, ...
[...] ethertype IPv4 (0x0800), length 98: 192.168.42.1 > 192.168.42.2: ICMP echo request, ...
[...] ethertype IPv4 (0x0800), length 98: 192.168.42.2 > 192.168.42.1: ICMP echo reply, ...
```

Traffic captured on `lan1.10` doesn't show any VLAN tag, as this interface will expose **untagged** traffic to the user.

Exit `tcpdump` with `ctrl-c`, and now dump the traffic going through `lan1` :

```
$ tcpdump -n -e -i lan1
```

```
[...] ethertype 802.1Q (0x8100), length 102: vlan 10, p 0, ethertype IPv4 (0x0800), 192.168.42
.1 ...
[...] ethertype 802.1Q (0x8100), length 102: vlan 10, p 0, ethertype IPv4 (0x0800), 192.168.42
.2 ...
[...] ethertype 802.1Q (0x8100), length 102: vlan 10, p 0, ethertype IPv4 (0x0800), 192.168.42
.1 ...
[...] ethertype 802.1Q (0x8100), length 102: vlan 10, p 0, ethertype IPv4 (0x0800), 192.168.42
.2 ...
```

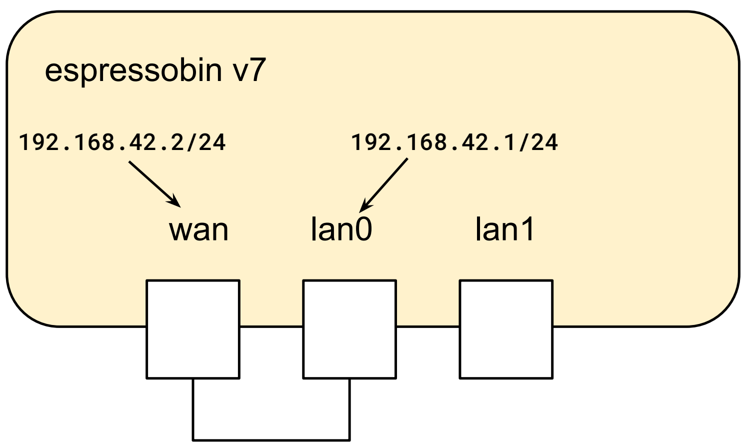
You can now notice that traffic going through `lan1` contains a **802.1Q** tag with the VLAN id of `10`. The frames are also 4-bytes longer, which corresponds to the length of a 802.1Q tag.

Looping back

Let's take a step back from VLANs for now, and focus on the other interfaces of the espressobin. Clean-up the previous part's setup by removing the vlan interface :

```
$ ip link del lan1.10
```

We'll try to perform a simple test by looping back the `wan` and `lan0` interface :



Connect `wan` and `lan0` together with one of the provided Ethernet cables, and configure both interfaces with the `ip` command, using the addresses indicated in the above schematic.

Check that your link parameters are sensible with `ethtool`, and that you can send simple ICMP traffic with `ping`.

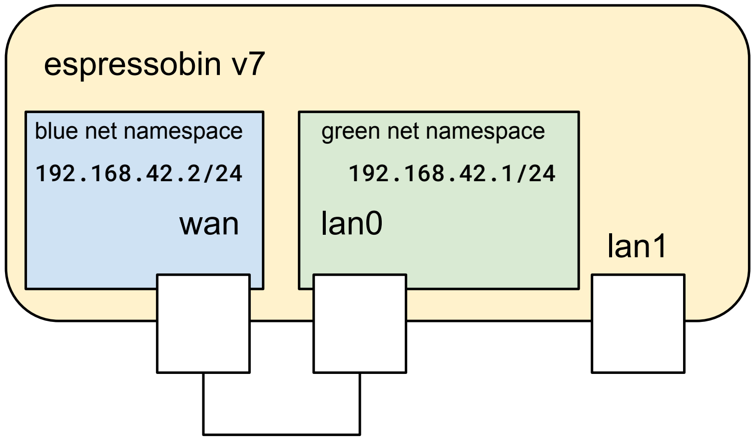
You will see the ping command working. However, we are pinging a local address, so make sure there's traffic flowing through our loopback cable. One of the ways of checking this is to looks at the **hardware counters** exposed by our drivers. This is done with `ethtool -S <iface>` :

```
$ ethtool -S wan | grep tx_packets
```

Check that this counter increments when sending traffic with `ping`.

As you may have guessed, the counters aren't incrementing although we do see the ping going through. This is because the Linux Kernel sees we're pinging a local addresses, and responds to it without involving our hardware interfaces.

To circumvent that, let's use **Network Namespaces** to isolate our interfaces from one another :



We'll use the `blue` netns for the `wan` interface, and the `green` netns for `lan0` :

```
$ ip netns add blue
$ ip link set dev wan netns blue

$ ip netns add green
$ ip link set dev lan0 netns green
```

From that point on, you will no longer see the `wan` and `lan0` interfaces when running `ip link show`, as they are no longer in the **init namespace**. To interact with them, you need to execute the commands within the namespace :

`ip netns exec green <cmd>`

You will now need to re-configure each interface, as they were flushed when entering the new namespace :

```
$ ip netns exec blue ip link set wan up
$ ip netns exec blue ip address add 192.168.42.2/24 dev wan

$ ip netns exec green ip link set lan0 up
$ ip netns exec green ip address add 192.168.42.1/24 dev lan0
```

Now, try to ping one the `wan` interface from the `green` namespace :

```
$ ip netns exec green ping 192.168.42.2
```

Check that the counters on `wan` are incrementing :

```
$ ip netns exec blue ethtool -S wan | grep tx_packets
```

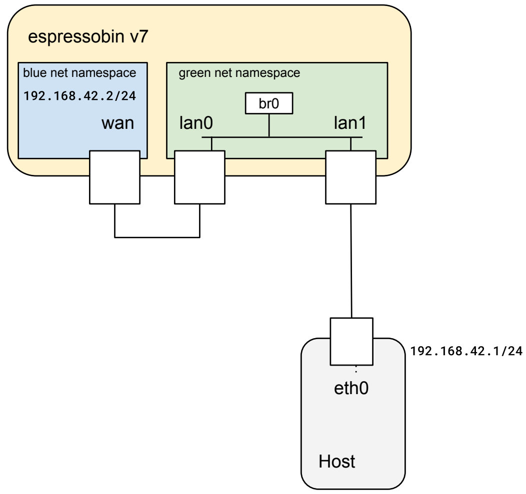
Bridging

Let's continue our quest to have the most convoluted setup with our espressobin.

We'll re-use what we have configured so-far with our namespaces, let's just flush the IP address from `lan0` :

```
$ ip netns exec green ip address flush lan0
```

We will create a **bridge** `lan0` and `lan1` so that we can use them as ports of a same **switch** :



First, let's create a new bridge device (in the green netns) :

```
$ ip netns exec green ip link add name br0 type bridge
$ ip netns exec green ip link set br0 up
```

Next, let's add `lan1` to the green netns :

```
$ ip link set lan1 netns green
$ ip netns exec green ip link set lan1 up
```

Finally, add both `lan0` and `lan1` to the bridge :

```
$ ip netns exec green ip link set dev lan0 master br0
$ ip netns exec green ip link set dev lan1 master br0
```

Make sure your Host's IP address is `192.168.42.2/24`, then try to ping it !

Congratulations !

Creating a new virtual netdev

Objective: learn how to interact with net devices in the kernel, and use netlink

Goals

- Create a very basic driver for a new Layer 2 network protocol
- Use `rtnl_link`
- Demonstrate stacked netdevice implementation

Bootlin LAN

In the next few labs, we will be implementing our custom Tagging protocol. It is similar to 802.1Q VLAN, with the following differences:

- Ethertype is ASCII ”BL”, or hex `0x424C`
- The BLAN id is encoded on 2 bytes, using the ascii representation of the tag's numerical value
- *e.g.* BLAN id 12 has the tag ”12” or hex `0x3132`
- Only one single BLAN can be added to a given trunk interface

MAC dst	MAC src	BLAN ethertype “BL”	tag : ascii	ethertype	Payload	FCS

Compiling and loading our new driver

For this lab, the code of the driver is located in `/home/$USER/networking-labs/target/blan`

This driver is out-of-tree, meaning that it is not compiled as part of the kernel, but as a separate kernel module.

To compile it, you need to use buildroot :

```
$ cd /home/$USER/networking-labs/buildroot
$ # Only re-compile the blan driver :
$ make bootlinlabs-blan-rebuild
$ # OR
$ # Only re-compile the blan driver and the full image
$ make bootlinlabs-blan-rebuild all
```

After doing so, you need to re-flash your sdcard using `dd`.

You can now boot your Espressobin, and insert the module corresponding to our driver :

```
insmod blan.ko
```

You should see your Hello World message being printed out.

Adding a new rtnl_link device type

Let's register our new link type to the `rtnl_link` family, by creating a new `rtnl_link_ops` object :


```
static struct rtnl_link_ops bootlinlan_link_ops = {
        .kind = "blan",
        .maxtype = IFLA_BLAN_MAX,
        .setup = bootlinlan_setup,
        .newlink = bootlinlan_newlink,
        .dellink = bootlinlan_dellink,
};
```

We will need some private data structure to be associated to our `struct net_device`, that will store context to be used in all our callback functions.

Create a new structure definition at the top of the file, for now empty :

```
struct bootlinlan_priv {
};
```

To make sure that the `struct net_device` associated to our new interface is allocated with enough room in its `net_device.priv` field, we can pass the size of the private data in the `rtnl_link` info :

```
static struct rtnl_link_ops bootlinlan_link_ops = {
...
       .priv_size = sizeof(struct bootlinlan_priv),
};
```

We now need to populate the 3 callback functions that were mentionned : `bootlinlan_setup`, `bootlinlan_newlink` and `bootlinlan_dellink`.

Take a look at the `struct rtnl_link_ops` definition to know the signature of these functions.

`bootlinlan_setup` will be called when the `rtnl_link` framework will allocate and initialize the `struct net_device` that will be created when running the `ip link` command.

There's 3 steps we need to take care of in the setup function :

- Specify the `struct net_device` information about the encapsulation (MTU, header size, etc). In our case, we can simply re-use the ones from a regular ethernet device by calling `ether_setup()`
- As we are not dealing with a driver for a real device (yet), `rtnl_link` will allocate a `struct net_device` for us. Therefore, we can make so that the networking stack also takes care of freeing the `struct net_device` for us when we are done with it. This is done by setting `dev->needs_free_netdev` to `true`
- Finally, all `struct net_device` objects need some NDOs (netdev ops) to be populated. Create a global variable of type `struct net_device_ops`, and pass its address into `dev->netdev_ops`. Populate the only required member of the ops, `.ndo_start_xmit()`. You can make a dummy function that does nothing but `return NETDEV_TX_OK;`.

Let's now focus on `bootlinlan_newlink`. This is were we'll focus most of our efforts in this lab.

This function takes as parameters :

- `struct net *src_net` : The network namespace in which the new device is created
- `struct net_device *dev` : The newly created `struct net_device`, that we want to configure
- `struct nlattr *tb[]` : A Netlink Attribute array containing attributes from the RTNL Family
- `struct nlattr *data[]` : A Netlink Attribute array contaning attributes specific to our link type
- `struct netlink_ext_ack *extack` : A Netlink Extended ACK object, for error reporting

In our case, the patched version of `iproute2` that is provided will :

- Place the parent's interface index into `tb[IFLA_LINK]`, as a u32 value

  – `ip link add link eth0 name blan0 type blan id 10`

Check that the IFLA_LINK value is provided (i.e that `tb[IFLA_LINK]` is not `NULL`). You can return `-EINVAL` if that's not the case.

You can retrieve the numerical values from attributes by using and `nla_get_u32()`.

The bootlinlan `id` is hardcoded in our drver, pick a value between 0 and 99, and store it in our local `bootlinlan_priv` struct for later use, make sure that you update the struct accordingly :

```
struct bootlinlan_priv {
        u16 id;
};
```

A pointer to your private data structure is stored in `net_device.priv` :

```
struct bootlinlan_priv *priv = netdev_priv(dev);
```

Then, you need to retrieve a pointer to the parent netdev. This will be used to configure our netdev. To get a netdev from its index, you can call `__dev_get_by_index()`. Take a look at its documentation to know what are the expected parameters.

Now that you have a pointer tothe parent netdev, we can configure our device :

- The `struct device` that backs our netdev is the same as the parent's. We can use `SET_NETDEV_DEV(dev, &parent_dev->dev);` to set it.
- We want to inherit the same MAC address as the parent. Use `eth_hw_addr_inherit()` for that.

We can finally register our device ! We need to call either `register_netdevice()`, or `register_netdev()`. Only one of them will work in our case :)

Once your code compiles, reboot your board, insert the module, and try to add a new `blan` device :

```
ip link add link lan0 name blan10 type blan
```

You should see your new device with :

```
ip link show
```

You can now populate the `bootlinlan_dellink` function, that unregisters the device. In our case, we need to use a special helper named `unregister_netdevice_queue()`, as we are currently holding the RTNL lock.

Configuring the stacked net devices

One final step is to notify the entire system that our `blan0` interface is stacked on `lan0`. To do this, add a call to `netdev_upper_dev_link()` after the registration of your netdev.

In the `bootlinlan_dellink`, the opposite is done by calling `netdev_upper_dev_unlink()`, however you'll notice that it takes as parameters both the parent device and our netdevice. So, let's add a `struct net_device` field in the `bootlinlan_priv` data structure :

```
struct bootlinlan_priv {
        struct net_device *lowerdev;
        u16 id;
};
```

Using Sockets in userspace

Objective: Learn the basics of socket programming

Goals

- Create a simple TCP client program
- Use tcpdump to visualise traffic
- Implement a simple packet dump program

Network configuration

In this lab, we'll setup a direct connection between the host and the target on the **192.168.42.0/24** subnet.

```
$ ip address flush lan0
$ ip link set lan0 up
$ ip address add 192.168.42.2/24 dev lan0
```

```
$ ip address flush <iface>
$ ip link set <iface> up
$ ip address add 192.168.42.1/24 dev <iface>
```

You can then run a simple `ping` test to make sure everything works :

```
$ ping 192.168.42.2
```

Simple TCP client

Let's start simple by creating a simple TCP client program, that will run on the Host, and connect to a server running on the targe. We will use the **netcat** program on the target, which provides a very simple implementation of TCP and UDP servers and clients.

Go in the lab3 host-side directory :

```
$ cd /home/$USER/networking-labs/host/lab3
```

In the `tcp_client.c` file, let's implement a very simple program to get familiar with the socket programming aspects. The program will take 2 parameters :

- The server's address, using the **dotted notation**
- The server's port

Start by creating your socket file descriptor, using the **socket** function. Use the `AF_xxx` family correspondig to the IPv4 address family. Use the `SOCK_xxx` type corresponding to TCP :

`int sockfd; sockfd = socket(AF_???, SOCK_???, 0);`

We pass a 0 as the last paraemeter as we won't be using any flags.

Next, create an object representing an IPv4 address : `struct sockaddr_in addr;`

You need to populate 3 fields within this struct :

- `addr.sin_family` : Use the `AF_xxx` family corresponding to IPv4

- `addr.sin_port` : The server's port. You can use `atoi()` to convert the user-pased ASCII string to an integer. This field must be specified in **network byte order**, use `htons()` to convert the port in the right endianness.
- `addr.sin_addr` : The server's IP address. You can use `inet_aton` to convert from dotted notation to the 32bits value.

Next, let's connect to the server. The `connect()` call must be made, but it expects a generic `struct sockaddr` as a parameter, which is subclassed by `struct sockaddr_in`, so we need to cast it back to the parent class :

```
connect(sockfd, (struct sockaddr *)&addr, sizeof(addr));
```

Finally, write the string "hi !" into the socket, using `write()` or `send()`.

Don't forget to `close()` the socket before terminating the program.

You can compile it with :

```
$ make tcp_client
```

Once your program looks good, you can start testing it !

Start the netcat TCP server on the Espressobin, listening on port 3000 :

```
$ nc -l -p 3000
```

And test your client on the host machine :

```
$ ./tcp_client 192.164.42.2 3000
```

You should see the string "hi !" printed on the Espressobin's console :)

Visualize traffic with TCPdump

Let's make sure that our data is indeed sent in TCP over IPv4. For that, we'll use the **tcpdump** tool.

First let's leave the netcat TCP server running in the background :

```
$ nc -l -k -p 3000 &
```

Now run **tcpdump** on the Espressobin, listening on `lan0`, with the following flags :

- `-n` : Print addresses instead of hostnames
- `-e` : Include Layer 2 information

```
$ tcpdump -n -e -i lan0
```

Send the message to the server from the client :

```
$ ./tcp_client 192.164.42.2 3000
```

You should see in the TCPDump output, on the target, the various packets exchanges during the establinshment of the TCP stream : SYN, SYN-ACK, ACK. Acknowledgments are represented with a `.` in the flags.

```
... Flags [S], ...
... Flags [S.], ...
... Flags [.], ...
```

To dump the content of the packets, you can use :

```
$ tcpdump -XX -n -e -i lan0
```

The Espressobin integrates a DSA switch, all traffic running on `lan0` goes through the **conduit interface** `eth0`. Try running the capture on `lan0` and `eth0`. Do you see any difference ?

RAW socket: Implementing our own custom TCPDUMP

TCPdump is based on the use of `AF_PACKET` sockets, which give access to raw Layer 2 frames. Let's write our very simple implementatin of tcpdump.

We will be running this program on the target, so move into the target-side of our lab folder :

```
$ cd /home/$USER/networking-labs/target/lab3
```

Open the `monitor.c` file and let's start implementing it. Start by opening a new socket, with the `AF_PACKET` family, `SOCK_RAW` type and using the `htons(ETH_P_ALL)` protocol to listen to everything.

To listen on a given interface, we need to `bind` the socket to the given interface.

As a reminder, `bind()` takes 3 parameters :

- `int sockfd` : The socket file descriptor
- `struct sockaddr *addr` : a sockaddr pointer. For `AF_PACKET`, we must use the `struct sockaddr_ll` subclass
- `socklen_t addrlen` : The size of our addres object, that is `sizeof(struct sockaddr_ll)`

The `stuct sockaddr_ll` is a generic structure representing a Layer 2 address :

```
struct sockaddr_ll {
    unsigned short sll_family;   /* Always AF_PACKET */
    unsigned short sll_protocol; /* Physical-layer protocol */
    int            sll_ifindex;  /* Interface number */
    unsigned short sll_hatype;   /* ARP hardware type */
    unsigned char  sll_pkttype;  /* Packet type */
    unsigned char  sll_halen;    /* Length of address */
    unsigned char  sll_addr[8];  /* Physical-layer address */
};
```

We need to populate :

- `sll_family` : `AF_PACKET`
- `sll_ifindex` : The interface index to listen to. Use `if_nametoindex()` to convert the interface name to its index

With these parameters constructed, call `bind()` on your socket.

You can now start reading traffic from your socket. To simplify a bit, let's read only the first 40 bytes of each packet.

In a `while` loop, call `recv` to read the first 40 bytes into a buffer, then display its content by calling the provided "hexdump" functin.

You can compile your code locally to test it :

```
$ make monitor
```

To install it on your target, you have to update your linux image :

```
$ # clean your host-compiled monitor program :
$ make clean
$ Go in your buildroot folder
$ cd ../../buildroot
$ make lab3-rebuild all
```

You can now re-flash your SDcard with the `dd` command, and reboot your Espressobin.

To run the monitor program :


```
$ cd lab3
$ ./monitor lan0
```

Creating a new virtual netdev

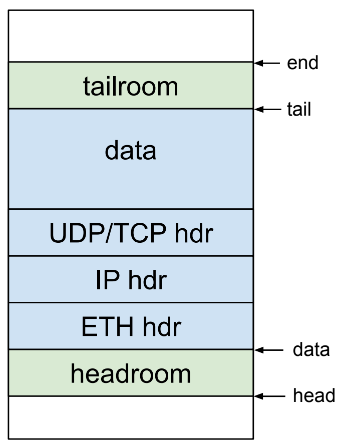
Objective: learn how to interact with net devices in the kernel, and use netlink

Goals

- Create a very basic driver for a new Layer 2 network protocol

Transmit path

Our driver will behave somewhat like what a 802.1Q driver would, that is we will receive a frame to be send (in the form of a `struct sk_buff` passed a parameter to our `.ndo_start_xmit` function), containing a fully-formed Ethernet frame :



Our custom tag needs to be inserted **before** the Ethernet header, that is, between the Layer 3 header and the Layer 2 header. We therefore need to move the Ethernet header down into the headroom of the SKB.

To guarantee that we have enough space in the headroom, we can add a constraint to our `struct net_device` to indicate that when `skb` are allocated with our device as a destination, there needs to be mode headroom for our tag.

This can be done by setting `dev->needed_headroom` to the size of our header, in our `bootlinlan_setup` function.

The kernel doesn't provide any helper to move the ethernet header around, so let's re-build our skb header in 3 steps :

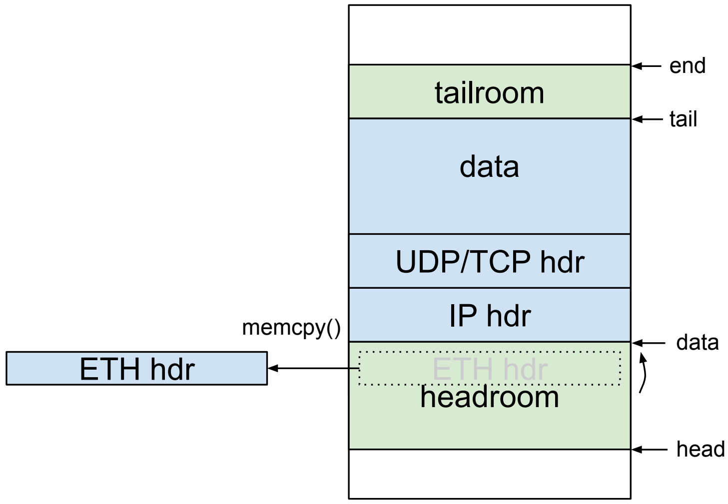
1. Save a copy fof the original ethernet header, and remove it
2. Insert our custom tag
3. Re-add the original ethernet header

Consume and copy the original ethernet header

In the `.ndo_start_xmit` callback of our driver, `skb->data` points at the beginning of the ethernet header. You can represent such a header by using `struct ethhdr` objects, which has the size `ETH_HLEN` (14 bytes).

Copy the `ETH_HLEN` bytes of `skb->data` into such an object.

Then, consume the first ETH_HLEN bytes of the payload section by calling the appropriate skb-manipulation helper.



Re-create the Layer 2 header

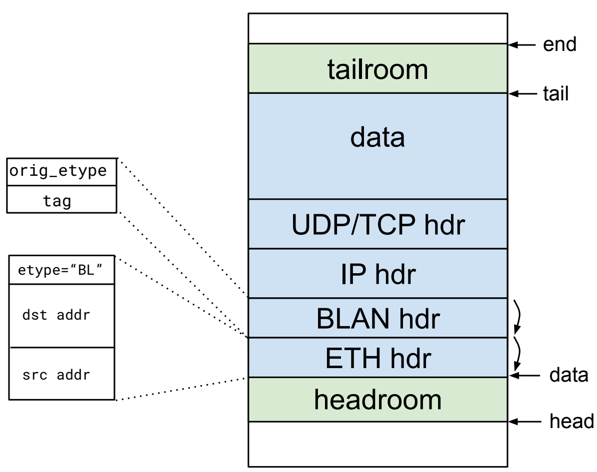
We now need to reconstruct the Layer 2 header. Using the approprate skb helper, re-increase the size of the skb by growing into the headroom the size of our blan header.

The helper will return a pointer to the newly grown section, you can directly assign it to your struct blan_hdr.

Fill-in the blan_hdr :

- The blan->etype should be the original ethernet header's Ethertype
- The blan->id should be the id associated to our netdev's priv data structure

You can now re-construct the Ethernet header. A useful helper for that is eth_header(), which does all the necessary skb manipulation (looking at its code may give you some clues :)).



Congratulations, it's now time to give your code a test :) Load the module, create the interface and assign it an IP address :

```
insmod blan.ko
ip link add link lan0 name blan10 type blan
```

```
ip address add 192.168.10.2/24 dev blan10
```

Launch `tcpdump` on your host machine, on the interface connected to your target :

```
tcpdump -n -e -i eth0
```

Start a `ping` from the target. It will not fully work yet, as we didn't implement the receive path, and the host machine can't understand our protocol :

```
ping 192.168.10.1
```


XDP - Using eBPF

Objective: learn eBPF programming using the XDP hook

Goals

- Compile our first eBPF program, and use it to drop everything
- Use eBPF maps through a userspace program
- Circumvent the hardware flaws of the armada3720 with XDP_REDIRECT

Compiling and loading our first XDP program

The Armada 3720 uses the mvneta driver, which has support for XDP upstream.

Let's first start by compiling and loading the simplest possible program, which will accept any incoming frame.

Let's do this work in our **buildroot overlay** directory :

```
$ cd /home/$USER/networking-labs/buildroot/overlay/root
$ mkdir xdp
$ cd xdp
```

Create a simple program named `xdp-pass.bpf.c` that always returns `XDP_PASS`, based on the following skeleton :

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int xdp_pass_prog(struct xdp_md *ctx)
{
/* Your code here */
}

char _license[] SEC("license") = "GPL";
```

Compile it to eBPF with `clang` :

```
clang -target bpf -g -O2 -c xdp-pass.bpf.c -o xdp-pass.bpf.o
```

Finally, we need to re-generate our linux image, so that it now contains our XDP program. We need to enable a few options in buildroot for that :

```
$ cd /home/$USER/networking-labs/buildroot
$ make menuconfig
```

Search with '/' the option "bpftool", enable it by pressing space. Then look for "elfutils", and also enable it. Move to the "Save" option, and hit "Quit" mutiple times until you get out of the menuconfig interface.

You can now re-generate the image :

```
$ make
```

Now re-flash your micro SDcard with the **dd** command, as explained in the setup lab.

Once your Espressbin has started, let's setup the network interface lan1

Filtering

Our first XDP test program will perform the simple task of dropping every packet that arrives onto the lan0 interface. This is not as simple as it sounds, as the Espressobin uses a DSA switch. This means that by default, the `eth0` port receives frames from ALL ports. It distinguises between the frames from each port by looking at the **DSA tag**. This will be covered in greater details in the next section of the training.

Each frame that arrives into our interfaces has the following layout :

```
.----.----.--------.--------.-----.----.---------
| DA | SA | 0xdada | 0x0000 | DSA | ET | Payload ...
'----'----'--------'--------'-----'----'---------
  6    6       2         2      4    2       N
```

This layout is called "Extended DSA", and includes an 8-byte in-between the MAC source address and the Ethertype.

The tag contains :

- a 2-bytes etherype `0xdada`
- 2 bytes of zeroes
- A 4-bytes DSA tag with the value :

  ```
  dsa_header[0] = (1 << 6) | tag_dev;
  dsa_header[1] = tag_port << 3;
  dsa_header[2] = 0;
  dsa_header[3] = 0;
  ```

In order to know the TAG id of a given interface, you need to look at the devicetree : `arch/arm64/boot/dts/marvell/armada-3720-espressobin.dtsi\#L182`

The Tag id is set by the `reg` value.

If we summarize, we only need to get the 8-byte tag, and recover the `tag_port` with

```
/* Skip the first 4 bytes of the EDSA tag*/

/* Check byte 1 of the DSA tag */
source_port = (dsa_header[1] >> 3) & 0x1f;
```

Create a new file in `buildroot/overlay/root/xdp` named `xdp-filter.bpf.c`, and use the following skeleton :

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int xdp_filter_prog(struct xdp_md *ctx)
{
        void *data_end = (void *)(long)ctx->data_end;
void *data = (void *)(long)ctx->data;

/* Grab the Ethernet header */
struct ethhdr *eth = data;

/* Check the size */
if (eth + 1 > data_end)
        return XDP_DROP;
```



```
/* Skip the 2 '0' bytes */
/* Get the next 4 bytes and check the source_port */

        return XDP_PASS;
}
char _license[] SEC("license") = "GPL";
```

Compile and load that program.

You should now be only able to use the LAN1 and WAN, the LAN0 interface should drop any incoming packet.

Investigating low level behaviour

Objective: Configure our testing setup for the labs

Goals

- Configure offloading and measure its impacts
- Analyse traffic and drops
- Use MQPrio to improve performances in specific setups
- Use traffic generation tools

Install the required packages

```
$ sudo apt install iperf3 dropwatch
```

Traffic generation with iperf3

Let's use `iperf3` to generate traffic between the Host and Target.

Run iperf3 in servermode (`-s`) and in the background (`-D`)

```
$ ip link set lan1 up
$ ip address add 192.168.42.2/24 dev lan1
$ iperf3 -s -D
```

Generate some traffic from your host :

```
$ iperf3 -c 192.168.42.2
```

By default, TCP traffic is sent. Let's now try UDP, using 400 bytes datagrams

```
$ iperf3 -c 192.168.42.2 -u -b 0 -l 400
```

The speed you are seeing is the speed at which the Host sends UDP to the Target.

To see the speed at which the target manages to receive the packets, you need to run the `iperf3` server in the foreground :

```
$ killall iperf3
$ iperf3 -s
```

Run the host-side iperf3 again. What do you see ?

Analysing performances

One way of investigating the ingress processing is by **profiling** the system.

Run the iperf3 UDP stream in the background, and start perf :

```
$ perf top
```

Can you identify the bottleneck ?

Using mqprio for traffic priorisation

When using a tool such as MQPrio, the process is more involved as you will need to classify your traffic, to indicate which packets are high-priority.

We will use VLANs for that purpose. We are going to create 2 flows between the host and target :

- An untagged flow, with low-priority traffic, on 192.168.42.0/24
- A tagged flow, with high-priority traffic, on 192.168.50.0/24

Create a VLAN interface on your host machine, with an **egress mapping** :

```
$ ip link add link <iface> name e.10 type vlan id 10 egress 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7
$ ip address add 192.168.50.1/24 dev e.10
$ ip address add 192.168.42.1/24 dev <iface>
```

Do the equivalent command on the Espressobin :

```
$ ip link add link lan1 name lan1.10 type vlan id 10 egress 0:0 1:1 2:2 3:3 4:4 5:5 6:6 7:7
$ ip address add 192.168.50.2/24 dev lan1.10
$ ip address add 192.168.42.2/24 dev lan1
```

Don't forget to put your interfaces UP !

Now, run the `iperf3` server on your Espressbin, in the background :

```
$ iperf3 -s -D
```

Let's now mark all traffic on `lan1.10` as priority 7 :

```
iptables -t mangle -A POSTROUTING -o lan1.100 -p udp -j CLASSIFY --set-class 0:7
iptables -t mangle -A POSTROUTING -o lan1.100 -p tcp -j CLASSIFY --set-class 0:7
```

Finally, let's configure **mqprio** :

```
tc qdisc add dev eth0 parent root handle 100 mqprio num_tc 2 \
map 0 0 0 0 0 0 0 1 \
queues 7@0 1@7 hw 1 mode channel shaper \
bw_rlimit min_rate 0 0 max_rate 50Mbit 1000Mbit
```

Now check that traffic on lan1 is rate-limited to 50Mbps and that traffic on lan1.100 goes at full speed !

