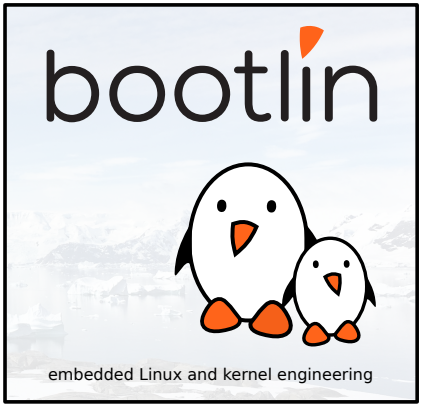
Ethernet switch support in the Linux kernel

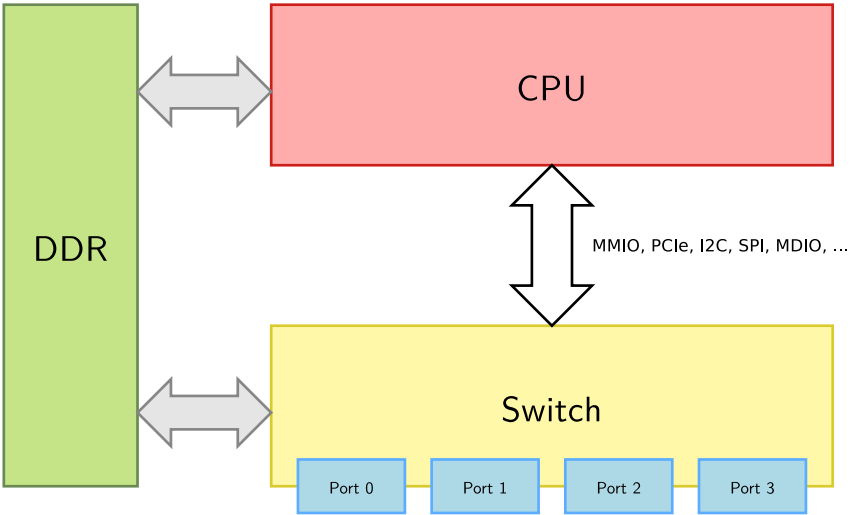
Alexandre Belloni
alexandre.belloni@bootlin.com



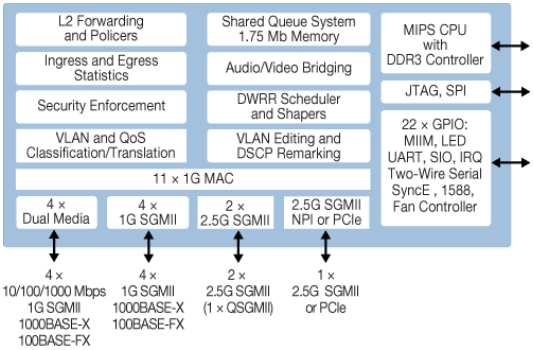
Alexandre Belloni

- Embedded Linux engineer at Bootlin
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
- Open-source contributor
  - Maintainer for the Linux kernel **RTC subsystem**
  - Co-Maintainer of **kernel support for Atmel ARM processors**

Switch


DDR
CPU
MMIO, PCIe, I2C, SPI, MDIO, ...
Switch
Port 0
Port 1
Port 2
Port 3

Microsemi VSC7514



- 500MHz MIPS CPU
- Usual controllers (UART, I2C, SPI)
- 2 MDIO controllers
- 10 port gigabit Ethernet switch
- 4 integrated PHYs
- Currently supported using an SDK running in userspace using UIO

Features

Usual hardware switch features include:

- Bridging
- STP
- MAC filtering
- IGMP snooping
- VLAN tagging/untagging

Linux switch support

- Switch ports are Linux network interfaces
- Standard Linux tools are used:
  - `ip`, `ifconfig` for interfaces
  - `ip`, `bridge`, `brctl` for bridging
  - Linux bonding for port trunks
- The switch can then accelerate what Linux can do in software
- `switchdev` is the Linux framework to offload features to the device

Switchdev

- Stateless framework, not using the device driver model
- `switchdev_ops` are attached to a `net_device` that has to be registered by the driver
- `switchdev_ops` implement offloading operations
- `switchdev_obj` abstracts objects (VLANS, MDB) to be used by the device

Front ports

Host mode

- Linux expects each port to be a network interface
- The VSC7514 doesn't have an Ethernet controller
- However, it is possible to extract or inject frames to/from the CPU
- Most of the initial configuration is to configure the ports to not forward frames and set up the CPU port for extraction and injection

Network devices, registration

- Each port is registered with `register_netdev` after setting the `struct net_device` members: `.netdev_ops`, `.ethtool_ops`, `.switchdev_ops`
- The interface MAC address is added to the switch MAC table
- If necessary, the phy is looked up and probed.

Network devices, `.ndo_open`

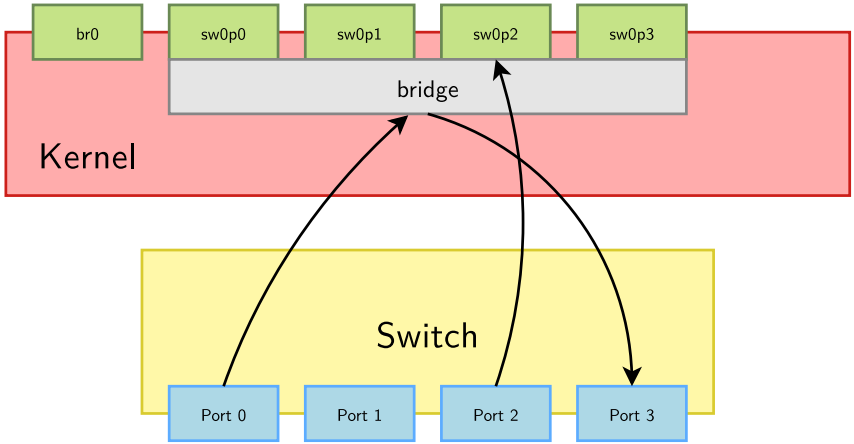
- `ifconfig sw0p0 up` or `ip link set dev sw0p0 up`
- Enable frame reception on the port and auto learning of MAC addresses
- Attach and start the phy.

Network devices, .ndo_start_xmit

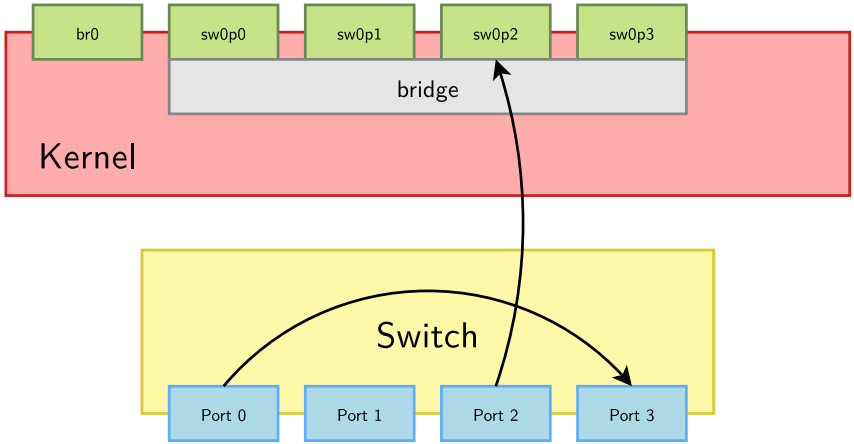
- The frames are injected on the CPU port and configured to be forwarded to the switch port
- There is a 128-bit header to specify what to do with the frame, in particular the port on which the frame has to be injected
- Frames can be transmitted using:
  - PIO with one register
  - DMA to DDR memory
  - DMA to 16 KB registers

Bridging and STP

Software bridging



Hardware offloading



Bridging

- Setting up a bridge can be done using `ip`:

```
ip link add name br0 type bridge
ip link set dev sw0p0 master br0
ip link set dev sw0p1 master br0
ip link set dev sw0p2 master br0
ip link set dev sw0p3 master br0
```

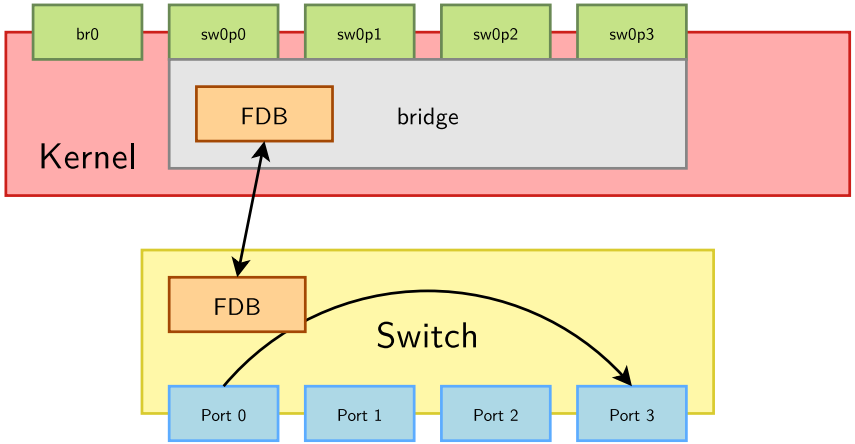
handling bridging

- Handling interface addition and removal is done through a netdevice notifier callback registered using `register_netdevice_notifier`
- The event is `NETDEV_CHANGEUPPER`. It is necessary to check the upper device is a bridge with `netif_is_bridge_master`

```
struct netdev_notifier_changeupper_info {
        struct netdev_notifier_info info; /* must be first */
        struct net_device *upper_dev; /* new upper dev */
        bool master; /* is upper dev master */
        bool linking; /* is the notification for link or unlink */
        void *upper_info; /* upper dev info */
};
```

- Check `info->linking` to discriminate between interface addition and removal

Forwarding database



FDB - Userspace

- Dumping the current bridge fdb table:

```
bridge fdb show
```

- Adding an FDB entry

```
bridge fdb add 00:00:05:00:01:00 dev sw0p0 static
```

FDB - kernel

- The bridge core and the switch FDB have to be kept in sync
- Handled using `.ndo_fdb_*` callbacks of the `net_device_ops` structure.
- At first, they were set to `switchdev_port_fdb_*` but they were removed in v4.14
- `.ndo_fdb_add` and `.ndo_fdb_del` are simple to implement, simply adds or removes a MAC entry
- `.ndo_fdb_dump` is more complicated as it has to handle netlink messaging. Taken mostly from DSA.
- This was necessary because the HW is not able to send interrupts when it learns a new MAC so the driver is not able to send an event to the bridge driver to maintain the FDB table.

  `call_switchdev_notifiers(SWITCHDEV_FDB_ADD_TO_BRIDGE, ...)`

Ageing

- Ageing time can be changed at setup using `ip`:

```
ip link set dev br0 type bridge ageing_time 1000
```

- or with `brctl`:

```
brctl setageing br0 1000
```

- The bridge core will call the `.switchdev_port_attr_set` callback of the registered `switchdev_ops`

switchdev_port_attr_set

```
int (*switchdev_port_attr_set)(struct net_device *dev,
                               const struct switchdev_attr *attr,
                               struct switchdev_trans *trans);
```

```
struct switchdev_attr {
        struct net_device *orig_dev;
        enum switchdev_attr_id id;
        u32 flags;
        void *complete_priv;
        void (*complete)(struct net_device *dev, int err, void *priv);
        union {
                struct netdev_phys_item_id ppid;        /* PORT_PARENT_ID */
                u8 stp_state;                           /* PORT_STP_STATE */
                unsigned long brport_flags;             /* PORT_BRIDGE_FLAGS */
                unsigned long brport_flags_support;     /* PORT_BRIDGE_FLAGS_SUPPORT */
                bool mrouter;                           /* PORT_MROUTER */
                clock_t ageing_time;                    /* BRIDGE_AGEING_TIME */
                bool vlan_filtering;                    /* BRIDGE_VLAN_FILTERING */
                bool mc_disabled;                       /* MC_DISABLED */
        } u;
};
```

Ageing

- `attr->id` is the attribute to set, for ageing, it will be `SWITCHDEV_ATTR_ID_BRIDGE_AGEING_TIME`
- `attr->u.ageing_time` holds the ageing time in jiffies
- `.switchdev_port_attr_set` is called twice to allow to change the configuration atomically. Use `switchdev_trans_ph_prepare(trans)` or `switchdev_trans_ph_commit(trans)` to know which step of the transaction this is.

STP

- Enabling STP is done with `brctl stp br0 on` or with `ip link set dev br0 type bridge stp_state 1`
- Handling STP is done through the `.switchdev_port_attr_set` callback
- `attr->id` will be `SWITCHDEV_ATTR_ID_PORT_STP_STATE`
- `attr->u.stp_state` hold the target STP state
- The various states are:
  - `BR_STATE_DISABLED`
  - `BR_STATE_LISTENING`
  - `BR_STATE_LEARNING`
  - `BR_STATE_FORWARDING`
  - `BR_STATE_BLOCKING`

Link aggregation

configuration

```
ip link add name aggr0 type bond
ip link set dev eth_yellow master aggr0
ip link set dev eth_blue master aggr0
```

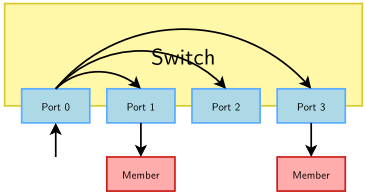
Link aggregation

- As for bridging, it uses the `NETDEV_CHANGEUPPER` in the netdevice notifier callback
- Check the upper device is a bond with `netif_is_lag_master`
- Check `info->linking` to discriminate between interface addition and removal

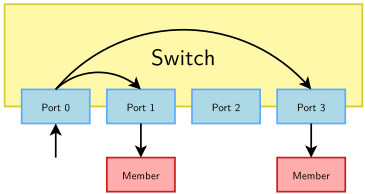
IGMP snooping

IGMP snooping

Without IGMP snooping



With IGMP snooping



IGMP snooping

- Linux can install MDBs so the switch avoids flooding multicast traffic on all ports.
- The switch needs to forward IGMP packets to the CPU. `netdev_for_each_mc_addr` will provide all the multicast addresses to be installed in the MAC table.
- The bridge core will call the `.switchdev_port_obj_add` callback of the registered `switchdev_ops`

switchdev_port_obj_add

```
int (*switchdev_port_obj_add)(struct net_device *dev,
                              const struct switchdev_obj *obj,
                              struct switchdev_trans *trans);
```

```
struct switchdev_obj {
        struct net_device *orig_dev;
        enum switchdev_obj_id id;
        u32 flags;
        void *complete_priv;
        void (*complete)(struct net_device *dev, int err, void *priv);
};
```

```
struct switchdev_obj_port_mdb {
        struct switchdev_obj obj;
        unsigned char addr[ETH_ALEN];
        u16 vid;
};

#define SWITCHDEV_OBJ_PORT_MDB(obj) \
        container_of(obj, struct switchdev_obj_port_mdb, obj)
```

installing MDBs

- `obj->id` will be `SWITCHDEV_OBJ_ID_PORT_MDB`
- Then cast to an mdb with `SWITCHDEV_OBJ_PORT_MDB()`
- The address and VLAN id are now available in `mdb->addr` and `mdb->vid`

VLAN filtering

configuration

```
ip link add name br0 type bridge
ip link set dev br0 type bridge vlan_filtering 1
ip link set dev sw0p0 master br0
ip link set dev sw0p1 master br0
ip link set dev sw0p2 master br0
ip link set dev sw0p3 master br0
bridge vlan add dev sw0p0 vid 1 pvid untagged
bridge vlan add dev sw0p1 vid 1
bridge vlan add dev sw0p2 vid 1
bridge vlan add dev sw0p3 vid 1
bridge vlan add dev sw0p0 vid 10
bridge vlan add dev sw0p1 vid 10 pvid untagged
bridge vlan add dev sw0p2 vid 20 pvid untagged
bridge vlan add dev sw0p3 vid 20
bridge vlan add dev sw0p0 vid 30
bridge vlan add dev sw0p1 vid 30
bridge vlan add dev sw0p2 vid 30
```

VLAN

- The bridge core will call the `.switchdev_port_obj_add` callback of the registered `switchdev_ops`
- This time, `obj->id` will be `SWITCHDEV_OBJ_ID_PORT_VLAN`

```
struct switchdev_obj_port_vlan {
        struct switchdev_obj obj;
        u16 flags;
        u16 vid_begin;
        u16 vid_end;
};

#define SWITCHDEV_OBJ_PORT_VLAN(obj) \
        container_of(obj, struct switchdev_obj_port_vlan, obj)
```

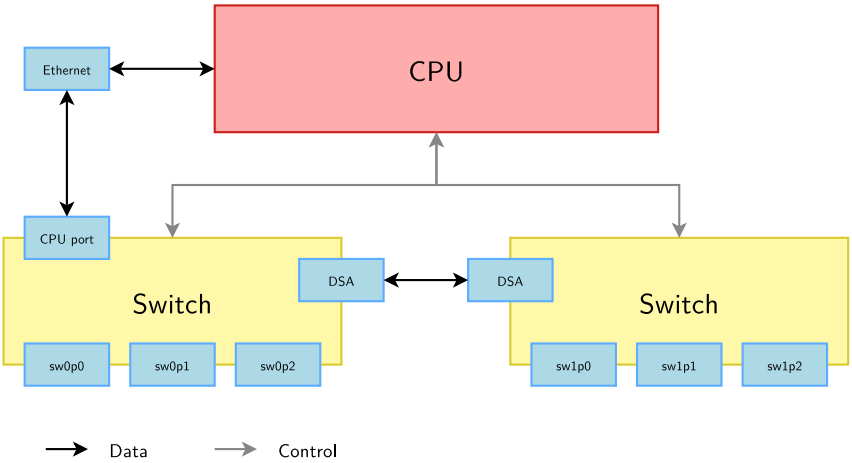
VLAN

- Cast to a `switchdev_obj_port_vlan` with `SWITCHDEV_OBJ_PORT_VLAN`
- All VLAN ids to install are from `vid_begin` to `vid_end`
- `flags` will be a combination of:

```
#define BRIDGE_VLAN_INFO_MASTER        (1<<0) /* Operate on Bridge device as well */
#define BRIDGE_VLAN_INFO_PVID          (1<<1) /* VLAN is PVID, ingress untagged */
#define BRIDGE_VLAN_INFO_UNTAGGED      (1<<2) /* VLAN egresses untagged */
#define BRIDGE_VLAN_INFO_RANGE_BEGIN   (1<<3) /* VLAN is start of vlan range */
#define BRIDGE_VLAN_INFO_RANGE_END     (1<<4) /* VLAN is end of vlan range */
#define BRIDGE_VLAN_INFO_BRENTRY       (1<<5) /* Global bridge VLAN entry */
```

DSA

DSA


Ethernet
CPU
CPU port
Switch
DSA
DSA
Switch
sw0p0
sw0p1
sw0p2
sw1p0
sw1p1
sw1p2
Data
Control

DSA

- Distributed Switch Architecture
- Handles chaining switches through Ethernet ports
- Handles the vendor specific switch tagging protocol
- Integrates nicely in the device model
- As a defined device tree binding

DSA vs switchdev

- Is the switch connected to the CPU through an Ethernet interface?
- Can that interface absorb all the traffic from the switch?
- Does the switch use switch tags?

Other challenges

- The switch can be used by the internal CPU using MMIO or by an external CPU using PCIe. Device tree is used to describe the switch when using MMIO. This becomes quite impractical when using the same switch connected through PCIe on x86. The current solution is to have an MFD driver registering all the necessary drivers as platform_devices.
- All the registers are packed in the register space, this complicates support for similar switches but with a different number of ports

Next steps

- Sending patches Upstream
- DMA
- PTP, IEEE1588 support
- QoS
- SyncE support
- Rework promiscuous support

Questions? Suggestions? Comments?

Alexandre Belloni
alexandre.belloni@bootlin.com

`https://bootlin.com/pub/conferences/2018/elc/`