Architecture

This document describes the **Distributed Switch Architecture (DSA)** subsystem design principles, limitations, interactions with other subsystems, and how to develop drivers for this subsystem as well as a TODO for developers interested in joining the effort.

Design principles

The Distributed Switch Architecture is a subsystem which was primarily designed to support Marvell Ethernet switches (MV88E6xxx, a.k.a Linkstreet product line) using Linux, but has since evolved to support other vendors as well.

The original philosophy behind this design was to be able to use unmodified Linux tools such as bridge, iproute2, ifconfig to work transparently whether they configured/queried a switch port network device or a regular network device.

An Ethernet switch is typically comprised of multiple front-panel ports, and one or more CPU or management port. The DSA subsystem currently relies on the presence of a management port connected to an Ethernet controller capable of receiving Ethernet frames from the switch. This is a very common setup for all kinds of Ethernet switches found in Small Home and Office products: routers, gateways, or even top-of-the rack switches. This host Ethernet controller will be later referred to as "master" and "cpu" in DSA terminology and code.

The D in DSA stands for Distributed, because the subsystem has been designed with the ability to configure and manage cascaded switches on top of each other using upstream and downstream Ethernet links between switches. These specific ports are referred to as "dsa" ports in DSA terminology and code. A collection of multiple switches connected to each other is called a "switch tree".

For each front-panel port, DSA will create specialized network devices which are used as controlling and data-flowing endpoints for use by the Linux networking stack. These specialized network interfaces are referred to as "slave" network interfaces in DSA terminology and code.

The ideal case for using DSA is when an Ethernet switch supports a "switch tag" which is a hardware feature making the switch insert a specific tag for each Ethernet frames it received to/from specific ports to help the management interface figure out:

- what port is this frame coming from
- what was the reason why this frame got forwarded
- how to send CPU originated traffic to specific ports

The subsystem does support switches not capable of inserting/stripping tags, but the features might be slightly limited in that case (traffic separation relies on Port-based VLAN IDs).

Note that DSA does not currently create network interfaces for the "cpu" and "dsa" ports because:

- the "cpu" port is the Ethernet switch facing side of the management controller, and as such, would create a duplication of feature, since you would get two interfaces for the same conduit: master netdev, and "cpu" netdev
- the "dsa" port(s) are just conduits between two or more switches, and as such cannot really be used as proper network interfaces either, only the downstream, or the top-most upstream interface makes sense with that model

Switch tagging protocols

DSA supports many vendor-specific tagging protocols, one software-defined tagging protocol, and a tag-less mode as well (`DSA_TAG_PROTO_NONE`).

The exact format of the tag protocol is vendor specific, but in general, they all contain something which:

- identifies which port the Ethernet frame came from/should be sent to
- provides a reason why this frame was forwarded to the management interface

All tagging protocols are in `net/dsa/tag_*.c` files and implement the methods of the `struct dsa_device_ops` structure, which are detailed below.

Tagging protocols generally fall in one of three categories:

1. The switch-specific frame header is located before the Ethernet header, shifting to the right (from the perspective of the DSA master's frame parser) the MAC DA, MAC SA, EtherType and the entire L2 payload.
2. The switch-specific frame header is located before the EtherType, keeping the MAC DA and MAC SA in place from the DSA master's perspective, but shifting the 'real' EtherType and L2 payload to the right.
3. The switch-specific frame header is located at the tail of the packet, keeping all frame headers in place and not altering the view of the packet that the DSA master's frame parser has.

A tagging protocol may tag all packets with switch tags of the same length, or the tag length might vary (for example packets with PTP timestamps might require an extended switch tag, or there might be one tag length on TX and a different one on RX). Either way, the tagging protocol driver must populate the `struct dsa_device_ops::needed_headroom` and/or `struct dsa_device_ops::needed_tailroom` with the length in octets of the longest switch frame header/trailer. The DSA framework will automatically adjust the MTU of the master interface to accommodate for this extra size in order for DSA user ports to support the standard MTU (L2 payload length) of 1500 octets. The `needed_headroom` and `needed_tailroom` properties are also used to

request from the network stack, on a best-effort basis, the allocation of packets with enough extra space such that the act of pushing the switch tag on transmission of a packet does not cause it to reallocate due to lack of memory.

Even though applications are not expected to parse DSA-specific frame headers, the format on the wire of the tagging protocol represents an Application Binary Interface exposed by the kernel towards user space, for decoders such as `libpcap`. The tagging protocol driver must populate the `proto` member of `struct dsa_device_ops` with a value that uniquely describes the characteristics of the interaction required between the switch hardware and the data path driver: the offset of each bit field within the frame header and any stateful processing required to deal with the frames (as may be required for PTP timestamping).

From the perspective of the network stack, all switches within the same DSA switch tree use the same tagging protocol. In case of a packet transiting a fabric with more than one switch, the switch-specific frame header is inserted by the first switch in the fabric that the packet was received on. This header typically contains information regarding its type (whether it is a control frame that must be trapped to the CPU, or a data frame to be forwarded). Control frames should be decapsulated only by the software data path, whereas data frames might also be autonomously forwarded towards other user ports of other switches from the same fabric, and in this case, the outermost switch ports must decapsulate the packet.

Note that in certain cases, it might be the case that the tagging format used by a leaf switch (not connected directly to the CPU) to not be the same as what the network stack sees. This can be seen with Marvell switch trees, where the CPU port can be configured to use either the DSA or the Ethertype DSA (EDSA) format, but the DSA links are configured to use the shorter (without Ethertype) DSA frame header, in order to reduce the autonomous packet forwarding overhead. It still remains the case that, if the DSA switch tree is configured for the EDSA tagging protocol, the operating system sees EDSA-tagged packets from the leaf switches that tagged them with the shorter DSA header. This can be done because the Marvell switch connected directly to the CPU is configured to perform tag translation between DSA and EDSA (which is simply the operation of adding or removing the `ETH_P_EDSA` EtherType and some padding octets).

It is possible to construct cascaded setups of DSA switches even if their tagging protocols are not compatible with one another. In this case, there are no DSA links in this fabric, and each switch constitutes a disjoint DSA switch tree. The DSA links are viewed as simply a pair of a DSA master (the out-facing port of the upstream DSA switch) and a CPU port (the in-facing port of the downstream DSA switch).

The tagging protocol of the attached DSA switch tree can be viewed through the `dsa/tagging` sysfs attribute of the DSA master:

```
cat /sys/class/net/eth0/dsa/tagging
```

If the hardware and driver are capable, the tagging protocol of the DSA switch tree can be changed at runtime. This is done by writing the new tagging protocol name to the same sysfs device attribute as above (the DSA master and all attached switch ports must be down while doing this).

It is desirable that all tagging protocols are testable with the `dsa_loop` mockup driver, which can be attached to any network interface. The goal is that any network interface should be capable of transmitting the same packet in the same way, and the tagger should decode the same received packet in the same way regardless of the driver used for the switch control path, and the driver used for the DSA master.

The transmission of a packet goes through the tagger's `xmit` function. The passed `struct sk_buff *skb` has `skb->data` pointing at `skb_mac_header(skb)`, i.e. at the destination MAC address, and the passed `struct net_device *dev` represents the virtual DSA user network interface whose hardware counterpart the packet must be steered to (i.e. `swp0`). The job of this method is to prepare the skb in a way that the switch will understand what egress port the packet is for (and not deliver it towards other ports). Typically this is fulfilled by pushing a frame header. Checking for insufficient size in the skb headroom or tailroom is unnecessary provided that the `needed_headroom` and `needed_tailroom` properties were filled out properly, because DSA ensures there is enough space before calling this method.

The reception of a packet goes through the tagger's `rcv` function. The passed `struct sk_buff *skb` has `skb->data` pointing at `skb_mac_header(skb) + ETH_ALEN` octets, i.e. to where the first octet after the EtherType would have been, were this frame not tagged. The role of this method is to consume the frame header, adjust `skb->data` to really point at the first octet after the EtherType, and to change `skb->dev` to point to the virtual DSA user network interface corresponding to the physical front-facing switch port that the packet was received on.

Since tagging protocols in category 1 and 2 break software (and most often also hardware) packet dissection on the DSA master, features such as RPS (Receive Packet Steering) on the DSA master would be broken. The DSA framework deals with this by hooking into the flow dissector and shifting the offset at which the IP header is to be found in the tagged frame as seen by the DSA master. This behavior is automatic based on the `overhead` value of the tagging protocol. If not all packets are of equal size, the tagger can implement the `flow_dissect` method of the `struct dsa_device_ops` and override this default behavior by specifying the correct offset incurred by each individual RX packet. Tail taggers do not cause issues to the flow dissector.

Due to various reasons (most common being category 1 taggers being associated with DSA-unaware masters, mangling what the master perceives as MAC DA), the tagging protocol may require the DSA master to operate in promiscuous mode, to receive all frames regardless of the value of the MAC DA. This can be done by setting the `promisc_on_master` property of the `struct dsa_device_ops`. Note that this assumes a DSA-unaware master driver, which is the norm.

Master network devices

Master network devices are regular, unmodified Linux network device drivers for the CPU/management Ethernet interface. Such a

driver might occasionally need to know whether DSA is enabled (e.g.: to enable/disable specific offload features), but the DSA subsystem has been proven to work with industry standard drivers: `e1000e, mv643xx_eth` etc. without having to introduce modifications to these drivers. Such network devices are also often referred to as conduit network devices since they act as a pipe between the host processor and the hardware Ethernet switch.

Networking stack hooks

When a master netdev is used with DSA, a small hook is placed in the networking stack is in order to have the DSA subsystem process the Ethernet switch specific tagging protocol. DSA accomplishes this by registering a specific (and fake) Ethernet type (later becoming `skb->protocol`) with the networking stack, this is also known as a `ptype` or `packet_type`. A typical Ethernet Frame receive sequence looks like this:

Master network device (e.g.: e1000e):

1. Receive interrupt fires:
    - receive function is invoked
    - basic packet processing is done: getting length, status etc.
    - packet is prepared to be processed by the Ethernet layer by calling `eth_type_trans`
2. net/ethernet/eth.c:

```
eth_type_trans(skb, dev)
        if (dev->dsa_ptr != NULL)
                -> skb->protocol = ETH_P_XDSA
```

3. drivers/net/ethernet/*:

```
netif_receive_skb(skb)
        -> iterate over registered packet_type
                -> invoke handler for ETH_P_XDSA, calls dsa_switch_rcv()
```

4. net/dsa/dsa.c:

```
-> dsa_switch_rcv()
        -> invoke switch tag specific protocol handler in 'net/dsa/tag_*.c'
```

5. net/dsa/tag_*.c:
    - inspect and strip switch tag protocol to determine originating port
    - locate per-port network device
    - invoke `eth_type_trans()` with the DSA slave network device
    - invoked `netif_receive_skb()`

Past this point, the DSA slave network devices get delivered regular Ethernet frames that can be processed by the networking stack.

Slave network devices

Slave network devices created by DSA are stacked on top of their master network device, each of these network interfaces will be responsible for being a controlling and data-flowing end-point for each front-panel port of the switch. These interfaces are specialized in order to:

- insert/remove the switch tag protocol (if it exists) when sending traffic to/from specific switch ports
- query the switch for ethtool operations: statistics, link state, Wake-on-LAN, register dumps...
- external/internal PHY management: link, auto-negotiation etc.

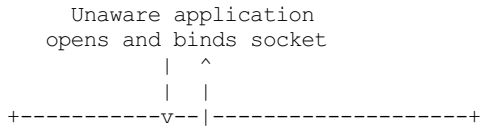
These slave network devices have custom net_device_ops and ethtool_ops function pointers which allow DSA to introduce a level of layering between the networking stack/ethtool, and the switch driver implementation.

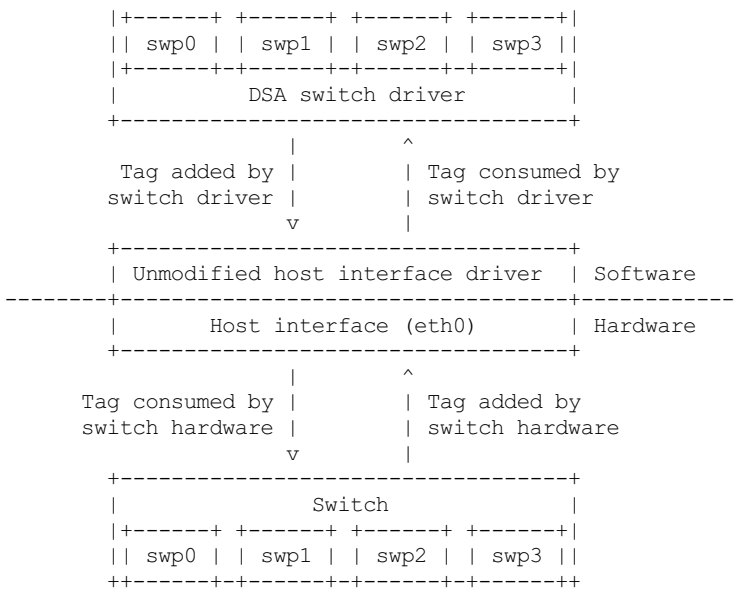
Upon frame transmission from these slave network devices, DSA will look up which switch tagging protocol is currently registered with these network devices, and invoke a specific transmit routine which takes care of adding the relevant switch tag in the Ethernet frames.

These frames are then queued for transmission using the master network device `ndo_start_xmit()` function, since they contain the appropriate switch tag, the Ethernet switch will be able to process these incoming frames from the management interface and delivers these frames to the physical switch port.

Graphical representation

Summarized, this is basically how DSA looks like from a network device perspective:





Slave MDIO bus

In order to be able to read to/from a switch PHY built into it, DSA creates a slave MDIO bus which allows a specific switch driver to divert and intercept MDIO reads/writes towards specific PHY addresses. In most MDIO-connected switches, these functions would utilize direct or indirect PHY addressing mode to return standard MII registers from the switch builtin PHYs, allowing the PHY library and/or to return link status, link partner pages, auto-negotiation results etc..

For Ethernet switches which have both external and internal MDIO busses, the slave MII bus can be utilized to mux/demux MDIO reads and writes towards either internal or external MDIO devices this switch might be connected to: internal PHYs, external PHYs, or even external switches.

Data structures

DSA data structures are defined in `include/net/dsa.h` as well as `net/dsa/dsa_priv.h`:

- `dsa_chip_data`: platform data configuration for a given switch device, this structure describes a switch device's parent device, its address, as well as various properties of its ports: names/labels, and finally a routing table indication (when cascading switches)
- `dsa_platform_data`: platform device configuration data which can reference a collection of dsa_chip_data structure if multiples switches are cascaded, the master network device this switch tree is attached to needs to be referenced
- `dsa_switch_tree`: structure assigned to the master network device under `dsa_ptr`, this structure references a dsa_platform_data structure as well as the tagging protocol supported by the switch tree, and which receive/transmit function hooks should be invoked, information about the directly attached switch is also provided: CPU port. Finally, a collection of dsa_switch are referenced to address individual switches in the tree.
- `dsa_switch`: structure describing a switch device in the tree, referencing a `dsa_switch_tree` as a backpointer, slave network devices, master network device, and a reference to the backing``dsa_switch_ops``
- `dsa_switch_ops`: structure referencing function pointers, see below for a full description.

Design limitations

Lack of CPU/DSA network devices

DSA does not currently create slave network devices for the CPU or DSA ports, as described before. This might be an issue in the following cases:

- inability to fetch switch CPU port statistics counters using ethtool, which can make it harder to debug MDIO switch connected using xMII interfaces
- inability to configure the CPU port link parameters based on the Ethernet controller capabilities attached to it: http://patchwork.ozlabs.org/patch/509806/
- inability to configure specific VLAN IDs / trunking VLANs between switches when using a cascaded setup

Common pitfalls using DSA setups

Once a master network device is configured to use DSA (dev->dsa_ptr becomes non-NULL), and the switch behind it expects a tagging protocol, this network interface can only exclusively be used as a conduit interface. Sending packets directly through this interface (e.g.: opening a socket using this interface) will not make us go through the switch tagging protocol transmit function, so the Ethernet switch on the other end, expecting a tag will typically drop this frame.

Interactions with other subsystems

DSA currently leverages the following subsystems:

- MDIO/PHY library: `drivers/net/phy/phy.c`, `mdio_bus.c`
- Switchdev:`net/switchdev/*`
- Device Tree for various of_* functions
- Devlink: `net/core/devlink.c`

MDIO/PHY library

Slave network devices exposed by DSA may or may not be interfacing with PHY devices (`struct phy_device` as defined in `include/linux/phy.h`), but the DSA subsystem deals with all possible combinations:

- internal PHY devices, built into the Ethernet switch hardware
- external PHY devices, connected via an internal or external MDIO bus
- internal PHY devices, connected via an internal MDIO bus
- special, non-autonegotiated or non MDIO-managed PHY devices: SFPs, MoCA; a.k.a fixed PHYs

The PHY configuration is done by the `dsa_slave_phy_setup()` function and the logic basically looks like this:

- if Device Tree is used, the PHY device is looked up using the standard "phy-handle" property, if found, this PHY device is created and registered using `of_phy_connect()`
- if Device Tree is used, and the PHY device is "fixed", that is, conforms to the definition of a non-MDIO managed PHY as defined in `Documentation/devicetree/bindings/net/fixed-link.txt`, the PHY is registered and connected transparently using the special fixed MDIO bus driver
- finally, if the PHY is built into the switch, as is very common with standalone switch packages, the PHY is probed using the slave MII bus created by DSA

SWITCHDEV

DSA directly utilizes SWITCHDEV when interfacing with the bridge layer, and more specifically with its VLAN filtering portion when configuring VLANs on top of per-port slave network devices. As of today, the only SWITCHDEV objects supported by DSA are the FDB and VLAN objects.

Devlink

DSA registers one devlink device per physical switch in the fabric. For each devlink device, every physical port (i.e. user ports, CPU ports, DSA links or unused ports) is exposed as a devlink port.

DSA drivers can make use of the following devlink features:

- Regions: debugging feature which allows user space to dump driver-defined areas of hardware information in a low-level, binary format. Both global regions as well as per-port regions are supported. It is possible to export devlink regions even for pieces of data that are already exposed in some way to the standard iproute2 user space programs (ip-link, bridge), like address tables and VLAN tables. For example, this might be useful if the tables contain additional hardware-specific details which are not visible through the iproute2 abstraction, or it might be useful to inspect these tables on the non-user ports too, which are invisible to iproute2 because no network interface is registered for them.
- Params: a feature which enables user to configure certain low-level tunable knobs pertaining to the device. Drivers may implement applicable generic devlink params, or may add new device-specific devlink params.
- Resources: a monitoring feature which enables users to see the degree of utilization of certain hardware tables in the device, such as FDB, VLAN, etc.
- Shared buffers: a QoS feature for adjusting and partitioning memory and frame reservations per port and per traffic class, in the ingress and egress directions, such that low-priority bulk traffic does not impede the processing of high-priority critical traffic.

For more details, consult `Documentation/networking/devlink/`.

Device Tree

DSA features a standardized binding which is documented in `Documentation/devicetree/bindings/net/dsa/dsa.txt`. PHY/MDIO library helper functions such as `of_get_phy_mode()`, `of_phy_connect()` are also used to query per-port PHY specific details: interface connection, MDIO bus location etc..

Driver development

DSA switch drivers need to implement a dsa_switch_ops structure which will contain the various members described below.

`register_switch_driver()` registers this dsa_switch_ops in its internal list of drivers to probe for. `unregister_switch_driver()` does the exact opposite.

Unless requested differently by setting the priv_size member accordingly, DSA does not allocate any driver private context space.

Switch configuration

- `tag_protocol`: this is to indicate what kind of tagging protocol is supported, should be a valid value from the `dsa_tag_protocol` enum
- `probe`: probe routine which will be invoked by the DSA platform device upon registration to test for the presence/absence of a switch device. For MDIO devices, it is recommended to issue a read towards internal registers using the switch pseudo-PHY and return whether this is a supported device. For other buses, return a non-NULL string
- `setup`: setup function for the switch, this function is responsible for setting up the `dsa_switch_ops` private structure with all it needs: register maps, interrupts, mutexes, locks etc.. This function is also expected to properly configure the switch to separate all network interfaces from each other, that is, they should be isolated by the switch hardware itself, typically by creating a Port-based VLAN ID for each port and allowing only the CPU port and the specific port to be in the forwarding vector. Ports that are unused by the platform should be disabled. Past this function, the switch is expected to be fully configured and ready to serve any kind of request. It is recommended to issue a software reset of the switch during this setup function in order to avoid relying on what a previous software agent such as a bootloader/firmware may have previously configured.

PHY devices and link management

- `get_phy_flags`: Some switches are interfaced to various kinds of Ethernet PHYs, if the PHY library PHY driver needs to know about information it cannot obtain on its own (e.g.: coming from switch memory mapped registers), this function should return a 32-bits bitmask of "flags", that is private between the switch driver and the Ethernet PHY driver in `drivers/net/phy/\*`.
- `phy_read`: Function invoked by the DSA slave MDIO bus when attempting to read the switch port MDIO registers. If unavailable, return 0xffff for each read. For builtin switch Ethernet PHYs, this function should allow reading the link status, auto-negotiation results, link partner pages etc..
- `phy_write`: Function invoked by the DSA slave MDIO bus when attempting to write to the switch port MDIO registers. If unavailable return a negative error code.
- `adjust_link`: Function invoked by the PHY library when a slave network device is attached to a PHY device. This function is responsible for appropriately configuring the switch port link parameters: speed, duplex, pause based on what the `phy_device` is providing.
- `fixed_link_update`: Function invoked by the PHY library, and specifically by the fixed PHY driver asking the switch driver for link parameters that could not be auto-negotiated, or obtained by reading the PHY registers through MDIO. This is particularly useful for specific kinds of hardware such as QSGMII, MoCA or other kinds of non-MDIO managed PHYs where out of band link information is obtained

Ethtool operations

- `get_strings`: ethtool function used to query the driver's strings, will typically return statistics strings, private flags strings etc.
- `get_ethtool_stats`: ethtool function used to query per-port statistics and return their values. DSA overlays slave network devices general statistics: RX/TX counters from the network device, with switch driver specific statistics per port
- `get_sset_count`: ethtool function used to query the number of statistics items
- `get_wol`: ethtool function used to obtain Wake-on-LAN settings per-port, this function may, for certain implementations also query the master network device Wake-on-LAN settings if this interface needs to participate in Wake-on-LAN
- `set_wol`: ethtool function used to configure Wake-on-LAN settings per-port, direct counterpart to set_wol with similar restrictions
- `set_eee`: ethtool function which is used to configure a switch port EEE (Green Ethernet) settings, can optionally invoke the PHY library to enable EEE at the PHY level if relevant. This function should enable EEE at the switch port MAC controller and data-processing logic
- `get_eee`: ethtool function which is used to query a switch port EEE settings, this function should return the EEE state of the switch port MAC controller and data-processing logic as well as query the PHY for its currently configured EEE settings
- `get_eeprom_len`: ethtool function returning for a given switch the EEPROM length/size in bytes
- `get_eeprom`: ethtool function returning for a given switch the EEPROM contents
- `set_eeprom`: ethtool function writing specified data to a given switch EEPROM
- `get_regs_len`: ethtool function returning the register length for a given switch
- `get_regs`: ethtool function returning the Ethernet switch internal register contents. This function might require user-land code in ethtool to pretty-print register values and registers

Power management

- `suspend`: function invoked by the DSA platform device when the system goes to suspend, should quiesce all Ethernet switch activities, but keep ports participating in Wake-on-LAN active as well as additional wake-up logic if supported
- `resume`: function invoked by the DSA platform device when the system resumes, should resume all Ethernet switch activities and re-configure the switch to be in a fully active state
- `port_enable`: function invoked by the DSA slave network device ndo_open function when a port is administratively brought up, this function should be fully enabling a given switch port. DSA takes care of marking the port with `BR_STATE_BLOCKING` if the port is a bridge member, or `BR_STATE_FORWARDING` if it was not, and propagating these changes down to the hardware
- `port_disable`: function invoked by the DSA slave network device ndo_close function when a port is administratively brought down, this function should be fully disabling a given switch port. DSA takes care of marking the port with

`BR_STATE_DISABLED` and propagating changes to the hardware if this port is disabled while being a bridge member

Bridge layer

- `port_bridge_join`: bridge layer function invoked when a given switch port is added to a bridge, this function should be doing the necessary at the switch level to permit the joining port from being added to the relevant logical domain for it to ingress/egress traffic with other members of the bridge.
- `port_bridge_leave`: bridge layer function invoked when a given switch port is removed from a bridge, this function should be doing the necessary at the switch level to deny the leaving port from ingress/egress traffic from the remaining bridge members. When the port leaves the bridge, it should be aged out at the switch hardware for the switch to (re) learn MAC addresses behind this port.
- `port_stp_state_set`: bridge layer function invoked when a given switch port STP state is computed by the bridge layer and should be propagated to switch hardware to forward/block/learn traffic. The switch driver is responsible for computing a STP state change based on current and asked parameters and perform the relevant ageing based on the intersection results
- `port_bridge_flags`: bridge layer function invoked when a port must configure its settings for e.g. flooding of unknown traffic or source address learning. The switch driver is responsible for initial setup of the standalone ports with address learning disabled and egress flooding of all types of traffic, then the DSA core notifies of any change to the bridge port flags when the port joins and leaves a bridge. DSA does not currently manage the bridge port flags for the CPU port. The assumption is that address learning should be statically enabled (if supported by the hardware) on the CPU port, and flooding towards the CPU port should also be enabled, due to a lack of an explicit address filtering mechanism in the DSA core.
- `port_bridge_tx_fwd_offload`: bridge layer function invoked after `port_bridge_join` when a driver sets `ds->num_fwd_offloading_bridges` to a non-zero value. Returning success in this function activates the TX forwarding offload bridge feature for this port, which enables the tagging protocol driver to inject data plane packets towards the bridging domain that the port is a part of. Data plane packets are subject to FDB lookup, hardware learning on the CPU port, and do not override the port STP state. Additionally, replication of data plane packets (multicast, flooding) is handled in hardware and the bridge driver will transmit a single skb for each packet that needs replication. The method is provided as a configuration point for drivers that need to configure the hardware for enabling this feature.
- `port_bridge_tx_fwd_unoffload`: bridge layer function invoken when a driver leaves a bridge port which had the TX forwarding offload feature enabled.

Bridge VLAN filtering

- `port_vlan_filtering`: bridge layer function invoked when the bridge gets configured for turning on or off VLAN filtering. If nothing specific needs to be done at the hardware level, this callback does not need to be implemented. When VLAN filtering is turned on, the hardware must be programmed with rejecting 802.1Q frames which have VLAN IDs outside of the programmed allowed VLAN ID map/rules. If there is no PVID programmed into the switch port, untagged frames must be rejected as well. When turned off the switch must accept any 802.1Q frames irrespective of their VLAN ID, and untagged frames are allowed.
- `port_vlan_add`: bridge layer function invoked when a VLAN is configured (tagged or untagged) for the given switch port. If the operation is not supported by the hardware, this function should return `-EOPNOTSUPP` to inform the bridge code to fallback to a software implementation.
- `port_vlan_del`: bridge layer function invoked when a VLAN is removed from the given switch port
- `port_vlan_dump`: bridge layer function invoked with a switchdev callback function that the driver has to call for each VLAN the given port is a member of. A switchdev object is used to carry the VID and bridge flags.
- `port_fdb_add`: bridge layer function invoked when the bridge wants to install a Forwarding Database entry, the switch hardware should be programmed with the specified address in the specified VLAN Id in the forwarding database associated with this VLAN ID. If the operation is not supported, this function should return `-EOPNOTSUPP` to inform the bridge code to fallback to a software implementation.

> **Note**
>
> VLAN ID 0 corresponds to the port private database, which, in the context of DSA, would be its port-based VLAN, used by the associated bridge device.

- `port_fdb_del`: bridge layer function invoked when the bridge wants to remove a Forwarding Database entry, the switch hardware should be programmed to delete the specified MAC address from the specified VLAN ID if it was mapped into this port forwarding database
- `port_fdb_dump`: bridge layer function invoked with a switchdev callback function that the driver has to call for each MAC address known to be behind the given port. A switchdev object is used to carry the VID and FDB info.
- `port_mdb_add`: bridge layer function invoked when the bridge wants to install a multicast database entry. If the operation is not supported, this function should return `-EOPNOTSUPP` to inform the bridge code to fallback to a software implementation. The switch hardware should be programmed with the specified address in the specified VLAN ID in the forwarding database associated with this VLAN ID.

> **Note**

VLAN ID 0 corresponds to the port private database, which, in the context of DSA, would be its port-based VLAN, used by the associated bridge device.

- `port_mdb_del`: bridge layer function invoked when the bridge wants to remove a multicast database entry, the switch hardware should be programmed to delete the specified MAC address from the specified VLAN ID if it was mapped into this port forwarding database.
- `port_mdb_dump`: bridge layer function invoked with a switchdev callback function that the driver has to call for each MAC address known to be behind the given port. A switchdev object is used to carry the VID and MDB info.

Link aggregation

Link aggregation is implemented in the Linux networking stack by the bonding and team drivers, which are modeled as virtual, stackable network interfaces. DSA is capable of offloading a link aggregation group (LAG) to hardware that supports the feature, and supports bridging between physical ports and LAGs, as well as between LAGs. A bonding/team interface which holds multiple physical ports constitutes a logical port, although DSA has no explicit concept of a logical port at the moment. Due to this, events where a LAG joins/leaves a bridge are treated as if all individual physical ports that are members of that LAG join/leave the bridge. Switchdev port attributes (VLAN filtering, STP state, etc) and objects (VLANs, MDB entries) offloaded to a LAG as bridge port are treated similarly: DSA offloads the same switchdev object / port attribute on all members of the LAG. Static bridge FDB entries on a LAG are not yet supported, since the DSA driver API does not have the concept of a logical port ID.

- `port_lag_join`: function invoked when a given switch port is added to a LAG. The driver may return `-EOPNOTSUPP`, and in this case, DSA will fall back to a software implementation where all traffic from this port is sent to the CPU.
- `port_lag_leave`: function invoked when a given switch port leaves a LAG and returns to operation as a standalone port.
- `port_lag_change`: function invoked when the link state of any member of the LAG changes, and the hashing function needs rebalancing to only make use of the subset of physical LAG member ports that are up.

Drivers that benefit from having an ID associated with each offloaded LAG can optionally populate `ds->num_lag_ids` from the `dsa_switch_ops::setup` method. The LAG ID associated with a bonding/team interface can then be retrieved by a DSA switch driver using the `dsa_lag_id` function.

IEC 62439-2 (MRP)

The Media Redundancy Protocol is a topology management protocol optimized for fast fault recovery time for ring networks, which has some components implemented as a function of the bridge driver. MRP uses management PDUs (Test, Topology, LinkDown/Up, Option) sent at a multicast destination MAC address range of 01:15:4e:00:00:0x and with an EtherType of 0x88e3. Depending on the node's role in the ring (MRM: Media Redundancy Manager, MRC: Media Redundancy Client, MRA: Media Redundancy Automanager), certain MRP PDUs might need to be terminated locally and others might need to be forwarded. An MRM might also benefit from offloading to hardware the creation and transmission of certain MRP PDUs (Test).

Normally an MRP instance can be created on top of any network interface, however in the case of a device with an offloaded data path such as DSA, it is necessary for the hardware, even if it is not MRP-aware, to be able to extract the MRP PDUs from the fabric before the driver can proceed with the software implementation. DSA today has no driver which is MRP-aware, therefore it only listens for the bare minimum switchdev objects required for the software assist to work properly. The operations are detailed below.

- `port_mrp_add` and `port_mrp_del`: notifies driver when an MRP instance with a certain ring ID, priority, primary port and secondary port is created/deleted.
- `port_mrp_add_ring_role` and `port_mrp_del_ring_role`: function invoked when an MRP instance changes ring roles between MRM or MRC. This affects which MRP PDUs should be trapped to software and which should be autonomously forwarded.

IEC 62439-3 (HSR/PRP)

The Parallel Redundancy Protocol (PRP) is a network redundancy protocol which works by duplicating and sequence numbering packets through two independent L2 networks (which are unaware of the PRP tail tags carried in the packets), and eliminating the duplicates at the receiver. The High-availability Seamless Redundancy (HSR) protocol is similar in concept, except all nodes that carry the redundant traffic are aware of the fact that it is HSR-tagged (because HSR uses a header with an EtherType of 0x892f) and are physically connected in a ring topology. Both HSR and PRP use supervision frames for monitoring the health of the network and for discovery of other nodes.

In Linux, both HSR and PRP are implemented in the hsr driver, which instantiates a virtual, stackable network interface with two member ports. The driver only implements the basic roles of DANH (Doubly Attached Node implementing HSR) and DANP (Doubly Attached Node implementing PRP); the roles of RedBox and QuadBox are not implemented (therefore, bridging a hsr network interface with a physical switch port does not produce the expected result).

A driver which is able of offloading certain functions of a DANP or DANH should declare the corresponding netdev features as indicated by the documentation at `Documentation/networking/netdev-features.rst`. Additionally, the following methods must be implemented:

- `port_hsr_join`: function invoked when a given switch port is added to a DANP/DANH. The driver may return `-EOPNOTSUPP` and in this case, DSA will fall back to a software implementation where all traffic from this port is sent to the

CPU.

- `port_hsr_leave`: function invoked when a given switch port leaves a DANP/DANH and returns to normal operation as a standalone port.

TODO

Making SWITCHDEV and DSA converge towards an unified codebase

SWITCHDEV properly takes care of abstracting the networking stack with offload capable hardware, but does not enforce a strict switch device driver model. On the other DSA enforces a fairly strict device driver model, and deals with most of the switch specific. At some point we should envision a merger between these two subsystems and get the best of both worlds.

Other hanging fruits

- allowing more than one CPU/management interface: http://comments.gmane.org/gmane.linux.network/365657