DSA switch configuration from userspace

The DSA switch configuration is not integrated into the main userspace network configuration suites by now and has to be performed manualy.

Configuration showcases

To configure a DSA switch a couple of commands need to be executed. In this documentation some common configuration scenarios are handled as showcases:

single port
Every switch port acts as a different configurable Ethernet port

bridge
Every switch port is part of one configurable Ethernet bridge

gateway
Every switch port except one upstream port is part of a configurable Ethernet bridge. The upstream port acts as different configurable Ethernet port.

All configurations are performed with tools from iproute2, which is available at https://www.kernel.org/pub/linux/utils/net/iproute2/

Through DSA every port of a switch is handled like a normal linux Ethernet interface. The CPU port is the switch port connected to an Ethernet MAC chip. The corresponding linux Ethernet interface is called the master interface. All other corresponding linux interfaces are called slave interfaces.

The slave interfaces depend on the master interface being up in order for them to send or receive traffic. Prior to kernel v5.12, the state of the master interface had to be managed explicitly by the user. Starting with kernel v5.12, the behavior is as follows:

- when a DSA slave interface is brought up, the master interface is automatically brought up.
- when the master interface is brought down, all DSA slave interfaces are automatically brought down.

In this documentation the following Ethernet interfaces are used:

eth0
the master interface

eth1
another master interface

lan1
a slave interface

lan2
another slave interface

lan3
a third slave interface

wan
A slave interface dedicated for upstream traffic

Further Ethernet interfaces can be configured similar. The configured IPs and networks are:

single port

- lan1: 192.0.2.1/30 (192.0.2.0 - 192.0.2.3)
- lan2: 192.0.2.5/30 (192.0.2.4 - 192.0.2.7)
- lan3: 192.0.2.9/30 (192.0.2.8 - 192.0.2.11)

bridge

- br0: 192.0.2.129/25 (192.0.2.128 - 192.0.2.255)

gateway

- br0: 192.0.2.129/25 (192.0.2.128 - 192.0.2.255)
- wan: 192.0.2.1/30 (192.0.2.0 - 192.0.2.3)

Configuration with tagging support

The tagging based configuration is desired and supported by the majority of DSA switches. These switches are capable to tag incoming and outgoing traffic without using a VLAN based configuration.

single port

```
# configure each interface
ip addr add 192.0.2.1/30 dev lan1
ip addr add 192.0.2.5/30 dev lan2
ip addr add 192.0.2.9/30 dev lan3

# For kernels earlier than v5.12, the master interface needs to be
# brought up manually before the slave ports.
ip link set eth0 up

# bring up the slave interfaces
ip link set lan1 up
ip link set lan2 up
ip link set lan3 up
```

bridge

```
# For kernels earlier than v5.12, the master interface needs to be
# brought up manually before the slave ports.
```

```
ip link set eth0 up

# bring up the slave interfaces
ip link set lan1 up
ip link set lan2 up
ip link set lan3 up

# create bridge
ip link add name br0 type bridge

# add ports to bridge
ip link set dev lan1 master br0
ip link set dev lan2 master br0
ip link set dev lan3 master br0

# configure the bridge
ip addr add 192.0.2.129/25 dev br0

# bring up the bridge
ip link set dev br0 up
```

gateway

```
# For kernels earlier than v5.12, the master interface needs to be
# brought up manually before the slave ports.
ip link set eth0 up

# bring up the slave interfaces
ip link set wan up
ip link set lan1 up
ip link set lan2 up

# configure the upstream port
ip addr add 192.0.2.1/30 dev wan

# create bridge
ip link add name br0 type bridge

# add ports to bridge
ip link set dev lan1 master br0
ip link set dev lan2 master br0

# configure the bridge
ip addr add 192.0.2.129/25 dev br0

# bring up the bridge
ip link set dev br0 up
```

Configuration without tagging support

A minority of switches are not capable to use a taging protocol (DSA_TAG_PROTO_NONE). These switches can be configured by a VLAN based configuration.

single port

The configuration can only be set up via VLAN tagging and bridge setup.

```
# tag traffic on CPU port
ip link add link eth0 name eth0.1 type vlan id 1
ip link add link eth0 name eth0.2 type vlan id 2
ip link add link eth0 name eth0.3 type vlan id 3

# For kernels earlier than v5.12, the master interface needs to be
# brought up manually before the slave ports.
ip link set eth0 up
ip link set eth0.1 up
ip link set eth0.2 up
ip link set eth0.3 up

# bring up the slave interfaces
ip link set lan1 up
ip link set lan2 up
ip link set lan3 up

# create bridge
ip link add name br0 type bridge

# activate VLAN filtering
ip link set dev br0 type bridge vlan_filtering 1

# add ports to bridges
ip link set dev lan1 master br0
ip link set dev lan2 master br0
ip link set dev lan3 master br0

# tag traffic on ports
bridge vlan add dev lan1 vid 1 pvid untagged
bridge vlan add dev lan2 vid 2 pvid untagged
bridge vlan add dev lan3 vid 3 pvid untagged

# configure the VLANs
ip addr add 192.0.2.1/30 dev eth0.1
ip addr add 192.0.2.5/30 dev eth0.2
ip addr add 192.0.2.9/30 dev eth0.3

# bring up the bridge devices
ip link set br0 up
```

bridge

```
# tag traffic on CPU port
ip link add link eth0 name eth0.1 type vlan id 1

# For kernels earlier than v5.12, the master interface needs to be
# brought up manually before the slave ports.
```

```
ip link set eth0 up
ip link set eth0.1 up

# bring up the slave interfaces
ip link set lan1 up
ip link set lan2 up
ip link set lan3 up

# create bridge
ip link add name br0 type bridge

# activate VLAN filtering
ip link set dev br0 type bridge vlan_filtering 1

# add ports to bridge
ip link set dev lan1 master br0
ip link set dev lan2 master br0
ip link set dev lan3 master br0
ip link set eth0.1 master br0

# tag traffic on ports
bridge vlan add dev lan1 vid 1 pvid untagged
bridge vlan add dev lan2 vid 1 pvid untagged
bridge vlan add dev lan3 vid 1 pvid untagged

# configure the bridge
ip addr add 192.0.2.129/25 dev br0

# bring up the bridge
ip link set dev br0 up
```

gateway

```
# tag traffic on CPU port
ip link add link eth0 name eth0.1 type vlan id 1
ip link add link eth0 name eth0.2 type vlan id 2

# For kernels earlier than v5.12, the master interface needs to be
# brought up manually before the slave ports.
ip link set eth0 up
ip link set eth0.1 up
ip link set eth0.2 up

# bring up the slave interfaces
ip link set wan up
ip link set lan1 up
ip link set lan2 up

# create bridge
ip link add name br0 type bridge

# activate VLAN filtering
ip link set dev br0 type bridge vlan_filtering 1
```

```
# add ports to bridges
ip link set dev wan master br0
ip link set eth0.1 master br0
ip link set dev lan1 master br0
ip link set dev lan2 master br0

# tag traffic on ports
bridge vlan add dev lan1 vid 1 pvid untagged
bridge vlan add dev lan2 vid 1 pvid untagged
bridge vlan add dev wan vid 2 pvid untagged

# configure the VLANs
ip addr add 192.0.2.1/30 dev eth0.2
ip addr add 192.0.2.129/25 dev br0

# bring up the bridge devices
ip link set br0 up
```

Forwarding database (FDB) management

The existing DSA switches do not have the necessary hardware support to keep the software FDB of the bridge in sync with the hardware tables, so the two tables are managed separately (`bridge fdb show` queries both, and depending on whether the `self` or `master` flags are being used, a `bridge fdb add` or `bridge fdb del` command acts upon entries from one or both tables).

Up until kernel v4.14, DSA only supported user space management of bridge FDB entries using the bridge bypass operations (which do not update the software FDB, just the hardware one) using the `self` flag (which is optional and can be omitted).

```
bridge fdb add dev swp0 00:01:02:03:04:05 self static
# or shorthand
bridge fdb add dev swp0 00:01:02:03:04:05 static
```

Due to a bug, the bridge bypass FDB implementation provided by DSA did not distinguish between `static` and `local` FDB entries (`static` are meant to be forwarded, while `local` are meant to be locally terminated, i.e. sent to the host port). Instead, all FDB entries with the `self` flag (implicit or explicit) are treated by DSA as `static` even if they are `local`.

```
# This command:
bridge fdb add dev swp0 00:01:02:03:04:05 static
# behaves the same for DSA as this command:
bridge fdb add dev swp0 00:01:02:03:04:05 local
# or shorthand, because the 'local' flag is implicit if 'static' is not
# specified, it also behaves the same as:
bridge fdb add dev swp0 00:01:02:03:04:05
```

The last command is an incorrect way of adding a static bridge FDB entry to a DSA switch using the bridge bypass operations, and works by mistake. Other drivers will treat an FDB entry added by the same command as `local` and as such, will not forward it, as opposed to DSA.

Between kernel v4.14 and v5.14, DSA has supported in parallel two modes of adding a bridge FDB entry to the switch: the bridge bypass discussed above, as well as a new mode using the `master` flag which installs FDB entries in the software bridge too.

```
bridge fdb add dev swp0 00:01:02:03:04:05 master static
```

Since kernel v5.14, DSA has gained stronger integration with the bridge's software FDB, and the support for its bridge bypass FDB implementation (using the `self` flag) has been removed. This results in the following changes:

```
# This is the only valid way of adding an FDB entry that is supported,
# compatible with v4.14 kernels and later:
bridge fdb add dev swp0 00:01:02:03:04:05 master static
# This command is no longer buggy and the entry is properly treated as
# 'local' instead of being forwarded:
bridge fdb add dev swp0 00:01:02:03:04:05
# This command no longer installs a static FDB entry to hardware:
bridge fdb add dev swp0 00:01:02:03:04:05 static
```

Script writers are therefore encouraged to use the `master static` set of flags when working with bridge FDB entries on DSA switch interfaces.

Affinity of user ports to CPU ports

Typically, DSA switches are attached to the host via a single Ethernet interface, but in cases where the switch chip is discrete, the hardware design may permit the use of 2 or more ports connected to the host, for an increase in termination throughput.

DSA can make use of multiple CPU ports in two ways. First, it is possible to statically assign the termination traffic associated with a certain user port to be processed by a certain CPU port. This way, user space can implement custom policies of static load balancing between user ports, by spreading the affinities according to the available CPU ports.

Secondly, it is possible to perform load balancing between CPU ports on a per packet basis, rather than statically assigning user ports to CPU ports. This can be achieved by placing the DSA masters under a LAG interface (bonding or team). DSA monitors this operation and creates a mirror of this software LAG on the CPU ports facing the physical DSA masters that constitute the LAG slave devices.

To make use of multiple CPU ports, the firmware (device tree) description of the switch must mark all the links between CPU ports and their DSA masters using the `ethernet` reference/phandle. At startup, only a single CPU port and DSA master will be used - the numerically first port from the firmware description which has an `ethernet` property. It is up to the user to configure the system for the switch to use other masters.

DSA uses the `rtnl_link_ops` mechanism (with a “dsa” `kind`) to allow changing the DSA master of a user port. The `IFLA_DSA_MASTER` u32 netlink attribute contains the ifindex of the master device that handles each slave device. The DSA master must be a valid candidate based on firmware node information, or a LAG interface which contains only slaves which are valid candidates.

Using iproute2, the following manipulations are possible:

```
# See the DSA master in current use
ip -d link show dev swp0
    (...)
    dsa master eth0

# Static CPU port distribution
ip link set swp0 type dsa master eth1
ip link set swp1 type dsa master eth0
ip link set swp2 type dsa master eth1
ip link set swp3 type dsa master eth0

# CPU ports in LAG, using explicit assignment of the DSA master
ip link add bond0 type bond mode balance-xor && ip link set bond0 up
ip link set eth1 down && ip link set eth1 master bond0
ip link set swp0 type dsa master bond0
ip link set swp1 type dsa master bond0
ip link set swp2 type dsa master bond0
ip link set swp3 type dsa master bond0
ip link set eth0 down && ip link set eth0 master bond0
ip -d link show dev swp0
    (...)
    dsa master bond0

# CPU ports in LAG, relying on implicit migration of the DSA master
ip link add bond0 type bond mode balance-xor && ip link set bond0 up
ip link set eth0 down && ip link set eth0 master bond0
ip link set eth1 down && ip link set eth1 master bond0
ip -d link show dev swp0
    (...)
    dsa master bond0
```

Notice that in the case of CPU ports under a LAG, the use of the `IFLA_DSA_MASTER` netlink attribute is not strictly needed, but rather, DSA reacts to the `IFLA_MASTER` attribute change of its present master (`eth0`) and migrates all user ports to the new upper of `eth0`, `bond0`. Similarly, when `bond0` is destroyed using `RTM_DELLINK`, DSA migrates the user ports that were assigned to this interface to the first physical DSA master which is eligible, based on the firmware description (it effectively reverts to the startup configuration).

In a setup with more than 2 physical CPU ports, it is therefore possible to mix static user to CPU port assignment with LAG between DSA masters. It is not possible to statically assign a user port towards a DSA master that has any upper interfaces (this includes LAG devices - the master must always be the LAG in this case).

Live changing of the DSA master (and thus CPU port) affinity of a user port is permitted, in order to allow dynamic redistribution in response to traffic.

Physical DSA masters are allowed to join and leave at any time a LAG interface used as a DSA master; however, DSA will reject a LAG interface as a valid candidate for being a DSA master unless it has at least one physical DSA master as a slave device.