Micrel Switch Usage Guide

Rev 1.0

October 30, 2014

Table of Contents

1 Revision History

Revision	Date	Summary of Changes
1.0	10/30/2014	Initial revision.

2 Introduction

This document describes how to modify the host network driver to support using the features of the Micrel switches. The supported switches are KSZ8463, KSZ8863/73, KSZ9566/7 and KSZ9897 equivalents. Most of the switch drivers are implemented as Linux SPI drivers, but I2C drivers are also available if needed.

3 Overview

The Micrel switch is usually connected to a MAC controller through a MII/RGMII interface. From the MAC controller point of view it can be just a PHY. Therefore the switch device is implemented as a Linux phy device to the network driver. As the switch is actually accessed through SPI or I2C rather than MIIM, the driver creates a virtual MDIO bus for the phy devices to be created. The first PHY 0 is for the whole switch, PHY 1 is for port 1, PHY 2 for port 2, and so on. The switch driver can also simulate the standard PHY registers if required, but that is generally not needed if the provided APIs are used.

The first step to make the Micrel switch work is to make sure the MII interface is connected correctly on the system board and the correct MII mode is used. The switch port connected with that MII interface is generally the last port of the switch, and is called the host port.

4 Software Driver Implementation

The default operation of the switch driver is to act as a PHY to the host network driver. The only functions provided then are to notify the host for link change and change all the port speed to a specific one. The driver can report individual port speed, but the one got from the network driver is the fixed one from the MII interface, as most network devices use that speed information to program the registers for proper operation.

If that is the only requirement needed the network driver does not need much modification. For some situations it requires each port to be treated as a network device. As the network devices are created in the host network driver, it requires modifications to that driver to support these special functions.

The simplest way is to make each port a network device. To act as a single unit a bridge needs to be created to bind those network ports.

Each port is treated as a network device for use in STP application, but it is still preferred to have a main device to run some other applications. In this case there is a main device for a whole

switch, and child devices for each port. A bridge is then created to run STP. There are some issues in this method as the received frame is passed to both child and parent devices. The driver tries to minimize these problems by passing only STP related frames to the child device.

There is a problem for Precision Time Protocol (PTP) support when using STP. As the PTP event messages cannot be blindly forwarded by a software bridge, the standard software operation does not work.

Some applications like to have the control to send packets to specific port but do not want a bridge. In this case a virtual VLAN device is created for each port so the application has an option to use the VLAN device or the main device. There is a performance hit to pass two frames to both VLAN and main devices, so the driver only does this for certain situations, like handling PTP message or MRP frames.

4.1 Distributed Switch Architecture

An alternate way is to use Distributed Switch Architecture (DSA), a network layer to utilize the ports as network devices. It requires almost no modifications to the host network driver. However, as the host network driver is impervious to a switch in being used, everything is forwarded by software and the switch hardware is not utilized.

Micrel switches use a tail tagging feature to know the received port and specify the destination port. Once enabled the tail tag is required in all frames to be transmitted. The DSA driver automatically adds the steps to prepare a frame with proper tail tag when sent from DSA ports, but if the host network driver needs to send frames through its main network device, it needs to call the special `tail_xmit` function to add a tail tag.

It requires modifying some kernel files related to DSA to add Micrel DSA support. The kernel configuration `CONFIG_NET_DSA_TAIL_TAG` is defined when Micrel DSA support is selected.

5 Host Network Driver Modifications

The Micrel switch driver is responsible for most of the network operations, but as it does not really transmit and receive frames it requires modifications to the host network driver to help the switch to achieve its goal.

The main switch code is in the file `ksz_sw_*.c`. The associated header is `ksz_sw_*.h`. The switch drivers can be under different names depending on the switch chips.

The Micrel switch driver is most likely provided as a Linux SPI driver because SPI access is the faster register access mode supported by the switch. One thing to note about the Linux SPI

driver model is that hardware register access can be interrupted by the kernel putting the access function to sleep. Therefore hardware register access cannot be used directly in interrupt context, and it requires workqueue to do most of the jobs.

The Micrel SPI switch driver can be run by itself as a driver module, but the driver code can also be included in a host network driver directly so that the network driver has more control over the switch functions. Either way the network driver has control over the switch through its provided APIs.

As the Micrel switch implementation and DSA support share some common code it is better to define this conditional to refer to either one:

```
#if defined(CONFIG_MICREL_SWITCH) || defined(CONFIG_NET_DSA_TAG_TAIL)
#define HAVE_MICREL_SWITCH
#endif
```

To support hardware STP operation the appropriate switch specific STP configuration needs to be enabled in the Linux kernel configurations first. Then the conditional `CONFIG_KSZ_STP` needs to be defined in the host network driver. The kernel bridge private header needs to be included.

```
#ifdef CONFIG_MICREL_SWITCH
#if defined(CONFIG_MICREL_KSZ8463_STP) ||
       defined(CONFIG_MICREL_KSZ8863_STP) ||
       defined(CONFIG_MICREL_KSZ9897_STP)
#define CONFIG_KSZ_STP
#endif

#ifdef CONFIG_KSZ_STP
#include <../net/bridge/br_private.h>
#endif
```

The file `ksz_common.c` needs to be included. It contains some common functions used by Micrel drivers.

If the entire switch driver code is included as indicated in the kernel configurations, the appropriate switch driver file is included. Otherwise only the switch header files are included.

```
#if defined(CONFIG_MICREL_KSZ8863_EMBEDDED)
#include “spi-ksz8863.c”
#elif defined(CONFIG_MICREL_KSZ8463_EMBEDDED)
#include “spi-ksz8463.c”
#elif defined(CONFIG_MICREL_KSZ9897_EMBEDDED)
#include “spi-ksz9897.c”
#elif defined(CONFIG_HAVE_KSZ8863)
#include “ksz8863.h”
#include “ksz_sw.h”
#elif defined(CONFIG_HAVE_KSZ8463)
#include “ks846xReg.h”
```

```
#include “ksz8463.h”
#include “ksz_sw.h”
#elif defined(CONFIG_HAVE_KSZ9897)
#include “ksz9897.h”
#include “ksz_sw_9897.h”
#endif
```

For DSA the number of ports has to be defined manually as it is not automatically determined for now.

```
#ifdef CONFIG_NET_DSA_TAG_TAIL
#define MICREL_MAX_PORTS  2

#ifdef CONFIG_HAVE_KSZ9897
#undef MICREL_MAX_PORTS
#define MICREL_MAX_PORTS  6
#endif

#include “setup_dsa.c”
#endif
```

If the switch driver file is not included, then the header files `ksz_sw_phy.h` and `ksz_spi_net.h` need to be included.

```
#if defined(HAVE_MICREL_SWITCH) && !defined(CONFIG_MICREL_SWITCH_EMBEDDED)
#include “ksz_sw_phy.h”
#include “ksz_spi_net.h”
#endif
```

Note the `ksz_spi_net.h` file contains the network driver private data structure. The host network driver needs to modify its private data structure to integrate the variables defined in the `dev_priv` structure. For simplicity it is referred to `dev_priv` as the network device private data structure but the host network driver may use other names.

The file `ksz_sw_sysfs_*.c` can be included to provide user switch operations. More details to use those operations are in the ***Micrel Switch Reference Guide*** or ***Micrel Switch Application Notes***.

```
#if defined(HAVE_MICREL_SWITCH) && !defined(CONFIG_MICREL_SWITCH_EMBEDDED)

#define USE_MIB
#if defined(CONFIG_HAVE_KSZ9897)
#include “ksz_sw_sysfs_9897.c”
#else
#include “ksz_sw_sysfs.c”
#endif
```

The ksz_sw_sysfs structure needs to be defined somewhere. During host network driver initialization the init_sw_sysfs function can be called to setup switch sysfs operations. When the driver exits the procedure exit_sw_sysfs should be called to release resources.

When the switch driver is run as a module it already provides sysfs operations. The file location is at /sys/bus/spi/devices/spi0.0/sw/. For network device the location is at /sys/class/net/eth0/sw/.

The Micrel switch driver stores everything related to the switch in the ksz_sw structure. This is defined inside the switch driver. For the host network driver to access this structure it needs to go through the Linux PHY device model. The switch driver exposes each external port of the switch as a PHY device and creates a MDIO bus named "spi_mii.0." The PHY id 0 operates over the whole switch, while PHY id 1 is port 1 and so on. To check whether the switch driver is started correctly and so has created a MDIO bus, the host network driver needs to called the kernel phy_attach function to retrieve the main PHY device.

```
struct ksz_sw *check_avail_switch(void)
{
	char phy_id[MII_BUS_ID_SIZE];
	char bus_id[MII_BUS_ID_SIZE];
	struct net_device netdev;
	struct ksz_sw *sw = NULL;
	struct phy_device *phydev = NULL;

	snprintf(bus_id, MII_BUS_ID_SIZE, "spi_mii.%d", 0);
	snprintf(phy_id, MII_BUS_ID_SIZE, PHY_ID_FMT, bus_id, 0);
	phydev = phy_attach(&netdev, phy_id, 0, PHY_INTERFACE_MODE_MII);
	if (!IS_ERR(phydev)) {
		struct phy_priv *phydata = phydev->priv;

		sw = phydata->port.sw;
		phy_detach(phydev);
	}
	return sw;
}
```

The switch instance can be retrieved with above code and used throughout the host network driver to access the switch functions.

Other places to add Micrel switch code are described in the sections below.

For DSA support only the file setup_dsa.c needs to be included and the micrel_switch_init function called to initialize the DSA system. The parameters supplied are the main network device and phy device.

```
micrel_switch_init(&micrel_switch_plat_data, NO_IRQ, &dev->dev, &phydev->bus->dev);
```

6 Switch APIs

6.1 Data Structures and Definitions

The switch driver uses the following defined data types in its data structures and functions:

Data types	
u8	8-bit unsigned value.
S16	16-bit signed value.
u16	16-bit unsigned value.
u32	32-bit unsigned value.
s64	64-bit signed value.
u64	64-bit unsigned value.

The macro SW_D sometimes is used to represent the switch's default register access width: 8-bit, 16-bit, or 32-bit.

Only the data fields related to special switch operations are described in the following data structures:

struct ksz_mac_table	
u8 mac_addr[ETH_ALEN]	MAC address.
u16 vid	VLAN tag value.
u16 fid	Filter ID.
u32 ports	Port membership.
u8 override:1	Override field.
u8 use_fid:1	Use FID field.
u8 valid:1	Valid field.

This structure is used to store the static MAC table entry. There are 8 static MAC table entries in most of the Micrel switches such as KSZ8863. For PTP support extra entries are used for software manipulation.

struct ksz_alu_table	
u16 owner	Device ownership.
u8 forward	Forward rule.
u8 valid:1	Valid field.

This structure is used with the ksz_mac_table structure to specify the device ownership and forward rule of the specific MAC address.

Forward rule	
FWD_HOST_OVERRIDE	Frame is forwarded to host port with override.
FWD_HOST	Frame is forwarded to host port.
FWD_STP_DEV	Frame will be forwarded to STP device.
FWD_MAIN_DEV	Frame will be forwarded to main device.
FWD_VLAN_DEV	Frame will be forwarded to VLAN device.

struct ksz_vlan_table	
u16 vid	VLAN tag value.
u16 fid	Filter ID.
u32 member	Port membership.
u8 valid:1	Valid field.

This structure is used to store the VLAN table entry. There are 16 entries in the switches like KSZ8863.

struct ksz_port_cfg	
u16 vid	VLAN tag value.
u16 member	Port membership.
int stp_state	STP state.

This structure is used to store port configurations.

struct ksz_sw_info	
struct ksz_mac_table mac_table[MULTI_MAC_TABLE_ENTRIES]	Static MAC table entries.
int multi_net	Network multicast addresses used
int multi_sys	System multicast addresses used.
u8 blocked_rx[BLOCK_RX_ENTRIES] [ETH_LEN]	Blocked receive addresses.
int blocked_rx_cnt	Blocked receive addresses count.
struct ksz_vlan_table vlan_table[VLAN_TABLE_ENTRIES]	VLAN table entries.
struct ksz_port_cfg port_cfg[TOTAL_PORT_NUM]	Port configurations.
u8 br_addr[ETH_ALEN]	Bridge MAC address.
u8 mac_addr[ETH_ALEN]	Switch MAC address.
u8 member	Current port membership.
u8 stp	STP port membership.
u8 stp_down	STP port down membership.
u8 fwd_ports	Number of ports to forward in STP operation.

This structure is used to store the hardware switch information.

struct ksz_port_info	
uint state	Connection status.
uint tx_rate	Transmit rate.
u8 duplex	Duplex mode.
u8 port_id	Port index.
u8 mac_addr[ETH_ALEN]	Port MAC address.

This structure is used to store the external port information.

struct ksz_sw_net_ops	
void setup_special	
void setup_dev	

void start	
int stop	
void open_dev	
void open_port	
void close_port	
void close	
void open	
u8 set_mac_addr	
int get_tx_len	
void add_tail_tag	
int get_tail_tag	
struct sk_buff *check_tx	
struct net_device *rx_dev	
int match_pkt	
struct net_device *parent_rx	
int port_vlan_rx	
struct sk_buff *final_skb	
u8 get_port_state	
u8 get_state	
void set_state	
void set_multi	
int stp_rx	
int blocked_rx	
void monitor_ports	

This structure is used to provide function access to the switch. Three of the functions, `get_port_state`, `get_state`, and `set_state` need user implementation.

struct ksz_sw	
void *dev	Pointer to parent hardware device.
void *phydev	Pointer to PHY device interface.
struct ksz_sw_info *info	Pointer to switch information structure.
struct ksz_port_info	Port information.

port_info[SWITCH_PORT_NUM]	
struct net_device *netdev[TOTAL_PORT_NUM]	Pointer to network devices.
struct phy_device *phy[TOTAL_PORT_NUM]	Pointer to PHY devices.
int dev_offset	Indication of a switch associated network device.
int phy_offset	Indication of the port associated PHY device.
struct ksz_timer_info *monitor_timer_info	Timer information for monitoring ports.
struct work_struct *stp_monitor	Workqueue for STP monitoring.
struct ksz_sw_net_ops *net_ops	Network related switch function access.
u16 rx_ports	Bitmap of ports with receive enabled.
u16 tx_ports	Bitmap of ports with transmit enabled.
int dev_count	Number of network devices this switch supports.
u32 vlan_id	Used for the VLAN port forwarding feature.
u16 vid	Used for the VLAN port forwarding feature.
int multi_dev	Used to specify multiple devices mode.
int stp	Used to enable STP.
int fast_aging	Used to enable fast aging.
struct ptp_info ptp_hw	PTP data structure for used with PTP operation.

This structure is used to store the Micrel switch general information. Note the actual hardware switch information is stored in the `ksz_sw_info` structure.

struct ksz_port	
int first_port	Port index.
int mib_port_cnt	MIB port count.
int port_cnt	Port count.
struct ksz_sw *sw	Switch structure pointer.
struct ksz_port_info *linked	Linked port information pointer.

This structure is used to store the virtual port information. The virtual port can be a switch port, or the whole switch. In the case of whole switch the linked port information pointer is automatically assigned to the first switch port which has a link.

It is expected the host network driver has defined a variable called `promiscuous` to indicate whether the network controller should be in promiscuous mode.

struct ksz_ptp_ops	
void init	
void exit	
int stop	
struct ptp_msg *check_msg	
int update_msg	
void get_rx_tstamp	
void get_tx_tstamp	
int hwtstamp_ioctl	
int dev_req	
void proc_intr	
int drop_pkt	
void get_rx_info	
void set_tx_info	

This structure is used to provide function access to the PTP engine. The two functions, `get_clk_cnt`, and `test_access_time` can be implemented to help calculate the register access delay.

6.2 Switch Functions

All of the switch register access functions are accessed through the `reg` field in the switch structure. The standard switch functions are accessed through the `ops` field. The network related functions are accessed through the `net_ops` field. Only the network related functions are described here.

Samples of register access functions are:

```
sw->reg->r8
sw->reg->w8
sw->reg->r16
```

```
sw->reg->w16
sw->reg->r32
sw->reg->w32
```

Samples of standard functions are:

```
sw->ops->chk
sw->ops->cfg
```

Samples of network functions are:

```
sw->net_ops->open
sw->net_ops->close
```

6.2.1 sw_setup_special

```
void sw_setup_special ( struct ksz_sw *sw, int *port_cnt, int
*mib_port_cnt, int *dev_cnt );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`int *port_cnt`	Buffer to hold the switch port count.
	`int *mib_port_cnt`	Buffer to hold the switch MIB port count.
	`int *dev_cnt`	Buffer to hold the network device count.
Return		None.
Description		This procedure determines the features of the switch.

This procedure is used to determine the features and functions of the switch upon network driver initialization. There are two things to decide how to run the switch: whether STP is enabled and how the ports are used. The driver environment variable `multi_dev` selects the multiple devices mode to use, and the variable `stp` enables STP support. In that case the multiple devices mode is set automatically.

As the environmental variables can be set in different places it is necessary to set these variables in the switch instance using the OR command:

```
sw->multi_dev |= multi_dev;
sw->stp |= stp;
sw->fast_aging |= fast_aging;
```

For STP operation it is also necessary to supply the `get_port_state` function as explained later. Also the two functions `get_priv_state` and `set_priv_state` need to be defined.

```
sw->net_ops->get_state = get_priv_state;
```

```
sw->net_ops->set_state = set_priv_state;
```

The `port_cnt` variable indicates how many ports in the switch; the `mib_port_cnt`, how many ports with MIB counters information. Normally it is the same as `port_cnt`. The `dev_cnt` variable indicates how many network devices to be created. They all need to be initialized to 1. After the function call they will be updated to the proper numbers to be used in later network device creation.

```
dev_cnt = 1;
port_cnt = 1;
mib_port_cnt = 1;
sw->net_ops->setup_special(sw, &port_cnt, &mib_port_cnt, &dev_cnt);
```

6.2.2 sw_setup_dev

```
void sw_setup_dev ( struct ksz_sw *sw, struct net_device *dev,
char *dev_name, struct ksz_port *port, int i, int port_cnt, int
mib_port_cnt );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct net_device *dev`	Network device instance.
	`char *dev_name`	The first created network device name.
	`struct ksz_port *port`	Port instance.
	`int i`	Device index.
	`int port_cnt`	Port count.
	`int mib_port_cnt`	MIB port count.
Return		None.
Description		This procedure initializes the port instance.

This procedure initializes the port instance to proper port information such as port count and port index. The created network device is also associated with the port. As explained before, each network device has one port instance, in which one or several switch ports are associated.

```
dev_name[0] = '\0';
for (i = 0; i < dev_cnt; i++) {
       dev = alloc_etherdev(sizeof(struct dev_priv));
       priv = netdev_priv(dev);

       phy_addr = i + sw->phy_offset;
       snprintf(bus_id, MII_BUS_ID_SIZE, “spi_mii.%d”, 0);
       snprintf(phy_id, MII_BUS_ID_SIZE, PHY_ID_FMT, bus_id, phy_addr);
       priv->phydev = phy_attach(dev, phy_id, 0, sw->interface);
```

```
	priv->parent = sw->dev;
	priv->dev = dev;
	sw->net_ops->setup_dev(sw, dev, dev_name, &priv->port, i, port_cnt,
mib_port_cnt);

	if (!dev_name[0])
		strlcpy(dev_name, dev->name, IFNAMSIZ);
}
```

6.2.3 sw_start

```
void sw_start ( struct ksz_sw *sw, u8 *addr );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`u8 *addr`	MAC address of the host network device.
Return		None
Description		This procedures setups the switch.

This procedure setups the switch with proper operation depending on the features selected. It programs the switch source filter with the supplied MAC address to avoid forwarding its own frames upon receiving. It is normally called once in the network device start function. It is called by the `sw_open_dev` procedure so that procedure can be used instead.

When PTP driver is used the `ptp_start` procedure is called in this procedure.

6.2.4 sw_stop

```
int sw_stop ( struct ksz_sw *sw, int complete );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`int complete`	Flag to reset completely or not.
Return	`int`	The reset indication.
Description		This function resets the switch.

This function is used to reset the switch to its default operation. It is normally called in the network device stop function.

When PTP driver is used the `ptp_stop` function is called in this function.

6.2.5 sw_open_dev

```
void sw_open_dev ( struct ksz_sw *sw, struct net_device *dev, u8
*addr );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`u8 *addr`	MAC address of the host network device.
Return		None.
Description		This procedures setups the switch.

This procedure operates the same as the `sw_start` procedure in addition to calling the `sw_init_mib` procedure to initialize the MIB counters. It should be called once in the network device start function.

6.2.6 sw_open_port

```
void sw_open_port ( struct ksz_sw *sw, struct net_device *dev,
struct ksz_port *port, u8 *state );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct net_device *dev`	Network device instance
	`struct ksz_port *port`	Port instance.
	`u8 *state`	The STP state of the port.
Return		None.
Description		This procedure setups the port instance.

This procedure setups the port instance and updates the port's STP state as necessary. The port link speed will be retrieved and updated as necessary. It is called in the network device start function after the `sw_open_dev` procedure.

```
priv = netdev_priv(dev);
sw->net_ops->open_port(sw, dev, &priv->port, &priv->state);
```

6.2.7 sw_close_port

```
void sw_close_port ( struct ksz_sw *sw, struct net_device *dev,
struct ksz_port *port );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct net_device *dev`	Network device instance.
	`struct ksz_port *port`	Port instance.
Return		None.
Description		This procedure shuts off the port if necessary.

This procedure is called in the network device stop function to shut off the port if necessary.

```
priv = netdev_priv(priv);
sw->net_ops->close_port(sw, dev, &priv->port);
```

6.2.8 sw_open

```
void sw_open ( struct ksz_sw *sw );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
Return		None.
Description		This procedure starts the switch monitor timer.

This procedure is called at the end of network device start function to start the switch monitor timer.

6.2.9 sw_close

```
void sw_close ( struct ksz_sw *sw );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
Return		None.
Description		This procedure stops the switch monitor timer.

This procedure is called at the end of the network device stop function to stop the switch monitor timer.

6.2.10 sw_set_mac_addr

```
u8 sw_set_mac_addr ( struct ksz_sw *sw, struct net_device *dev,
u8 promiscuous, int port );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct net_device *dev`	Network device instance.
	`u8 promiscuous`	The hardware promiscuous count.
	`int port`	The port index.
Return	`u8`	The updated hardware promiscuous count.
Description		This function updates the switch MAC address.

This function updates the switch when the host network device MAC address is changed. In multiple devices mode each port may have its own MAC address. In that case the host network device controller needs to be put in promiscuous mode for it to receive different unicast packets. The returned hardware promiscuous count will tell the host controller whether to turn on promiscuous mode or not.

```
priv = netdev_priv(dev);
promiscuous = hw->promiscuous;
promiscuous = sw->net_ops->set_mac_addr(sw, dev, promiscuous, priv->port.first_port);
if (promiscuous != hw->promiscuous) {
	hw->promiscuous = promiscuous;
	/* Turn of/off promiscuous mode. */
}
```

6.2.11 sw_get_tx_len

```
int sw_get_tx_len ( struct ksz_sw *sw, struct sk_buff *skb );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct sk_buff *skb`	Transmit socket buffer.
Return	`int`	The required length to send to switch.
Description		This function calculates the minimum length of the frame to send to the switch.

This function should be called at the beginning of the network device transmit function to find out the minimum length required to send the frame to the switch so that the network driver can allocate additional resource.

6.2.12 sw_add_tail_tag

```
void sw_add_tail_tag ( struct ksz_sw *sw, struct sk_buff *skb,
int port );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct sk_buff *skb`	Transmit socket buffer.
	`int port`	Transmit port.
Return		None.
Description		This procedures adds a tail tag to send the frame to specific ports of the switch.

This procedure is used to add the tail tag for sending the frame to specific ports..

When using DSA this procedure should be defined:

```
void net_add_tail_tag(struct sk_buff *skb, struct net_device *dev, int port)
{
	/* Get the sw pointer from private data or somewhere. */
	sw->net_ops->add_tail_tag(sw, skb, 1 << port);
}
```

6.2.13 sw_get_tail_tag

```
int sw_get_tail_tag ( u8 *trailer, int *port );
```

Parameters	`u8 *trailer`	The last byte of the frame.
	`int *port`	Buffer to to hold the receive port.
Return	`int`	Extra length that can be removed.
Description		This procedures returns the extra length used by the tail tag.

This function is used to find out how many extra bytes are used by the tail tag. Normally 1 byte is used, but for PTP message 4 additional bytes are used to store the receive timestamp. The receive port is also returned.

When using DSA this function should be defined:

```
int net_get_tail_tag(struct sk_buff *skb, struct net_device *dev, int *port)
{
	u8 *trailer;

	/* Get the sw pointer from private data or somewhere. */
```

```
	trailer = skb_tail_pointer(skb) - 1;
	return sw->net_ops->get_tail_tag(trailer, port);
}
```

6.2.14 sw_check_tx

```
struct sk_buff *sw_check_tx ( struct ksz_sw *sw, struct
net_device *dev, struct sk_buff *skb, struct ksz_port *priv );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct net_device *dev`	Network device instance.
	`struct sk_buff *skb`	Transmit socket buffer.
	`struct ksz_port *port`	Port instance.
Return	`struct sk_buff *skb`	NULL if the socket buffer is deleted.
Description		This functions checks transmit frame.

This function checks if the transmit frame should be dropped due to some switch limitations. The frame also may be modified to prepare sending it to the switch. It is called in the network device transmit function.

6.2.15 sw_final_skb

```
struct sk_buff *sw_final_skb ( struct ksz_sw *sw, struct
net_device *dev, struct sk_buff *skb, struct ksz_port *priv );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct net_device *dev`	Network device instance.
	`struct sk_buff *skb`	Transmit socket buffer.
	`struct ksz_port *port`	Port instance.
Return	`struct sk_buff *skb`	NULL if the socket buffer is deleted.
Description		This functions checks transmit frame.

This function calls `tail_xmit` function if DSA is used. Otherwise it calls `sw_check_tx` to prepare the transmit frame.

When PTP driver is used the `ptp_get_tx_tstamp` procedure is called to prepare the transmit timestamp..

```
priv = netdev_priv(dev);
```

```
skb = sw->net_ops->final_skb(sw, dev, skb, &priv->port);
if (!skb) {
	/* Free any resource allocated to transmit the frame. */
	return 0;
}
```

6.2.16 sw_rx_dev

```
struct net_device *sw_rx_dev ( struct ksz_sw *sw, u8 *data, u32
*len, int *tag, int *port );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`u8 *data`	Received frame data buffer
	`u32 *len`	
	`int *tag`	Buffer to hold the port tag.
	`int *port`	Buffer to hold the port index.
Return	`struct net_device *dev`	Returned network device instance.
Description		This function determines the received port.

This function determines the received port from the tail tag at the end of the frame. It checks whether the frame should be dropped depending on STP state of the port. After then it returns the proper network device instance associated with the received port. The supplied tag and port variables will be used in later function calls.

```
/* Allocate a socket buffer to retrieve the received frame data. */
dev = sw->net_ops->rx_dev(sw, skb->data, &len, &tag, &rx_port);
if (!dev) {
	dev_kfree_skb_irq(skb);
	return -ENODEV;
}
```

6.2.17 sw_match_pkt

```
int sw_match_pkt ( struct ksz_sw *sw, struct net_device *dev,
void **priv, int (*get_promiscuous)(void *ptr), int
(*match_multi)(void *ptr, u8 *data), struct sk_buff *skb, u8
h_promiscuous );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct net_device **dev`	Network device instance pointer.

	`void **priv`	Private data pointer.
	`int (*get_promiscuous) (void *ptr)`	Get private promiscuous state function.
	`int (*match_multi)(void *ptr, u8 *data)`	Match multicast frame function.
	`struct sk_buff *skb`	Received socket buffer.
	`u8 h_promiscuous`	The hardware promiscuous count.
Return	`int`	Indication of a match.
Description		This function matches incoming destination address.

This function is used to match unicast and multicast packets in case hardware promiscuous mode is enabled. A get private promiscuous mode function and a match multicast frame function should be provided.

```
static int priv_promiscuous(void *ptr)
{
	struct dev_priv *priv = ptr;

	return priv->promiscuous;
}

static int priv_match_multi(void *ptr, u8 *data)
{
	int i;
	struct dev_priv *priv = ptr;
	int drop = false;

	if (priv->multi_list_size)
		drop = true;
	for (i = 0; i < priv->multi_list_size; i++)
		if (!memcmp(data, priv->multi_list[i], ETH_ALEN) {
			drop = false;
			break;
		}
	return drop;
}

priv = netdev_priv(dev);
if (!sw->net_ops->match_pkt(sw, &dev, (void **) &priv, priv_promiscuous,
priv_match_multi, skb, hw->promiscuous)) {
	dev_kfree_skb_irq(skb);
	return 0;
}
/* dev may be changed to different one and then priv will be also updated. */
```

6.2.18 sw_parent_rx

```
void sw_parent_rx ( struct ksz_sw *sw, struct net_device *dev,
struct sk_buff *skb, int forward, struct net_device **pdev,
struct sk_buff **pskb );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct net_device *dev`	Network device instance.
	`struct sk_buff *skb`	Received socket buffer.
	`int forward`	Forward rules.
	`struct net_device **pdev`	Buffer to hold the parent network device.
	`struct sk_buff **pskb`	Buffer to hold the parent socket buffer.
Return		None.
Description		This procedure creates another copy of socket buffer if necessary.

This procedure is used to create another copy of socket buffer for the parent network device if necessary. It is called after the `sw_stp_rx` function for the `forward` variable to be updated first if STP is used.

```
int forward = 0;
struct net_device *parent_dev = NULL;
struct sk_buff *parent_skb = NULL;

sw->net_ops->parent_rx(sw, dev, skb, forward, &parent_dev, &parent_skb);
```

6.2.19 sw_port_vlan_rx

```
int sw_port_vlan_rx ( struct ksz_sw *sw, struct net_device *dev,
struct net_device *pdev, struct sk_buff *skb, int forward, int
tag, void *ptr, void (*rx_tstamp)(void *ptr, struct sk_buff *skb)
);
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct net_device *dev`	Network device instance.
	`struct net_device *pdev`	Parent network device instance.
	`struct sk_buff *skb`	Received socket buffer.
	`int forward`	Forward rules.
	`int tag`	Port tag.

	`void *ptr`	PTP pointer if used.
	`void (*rx_tstamp)(void *ptr, struct sk_buff *skb)`	PTP receive timestamp function if used.
Return	`int`	Indication a socket buffer is passed to a VLAN device.
Description		This function creates a copy of socket buffer to pass to a VLAN device if necessary.

This function checks whether a copy of received socket buffer needs to be passed to a VLAN device so that application receives it knows which port the packet is received. Some of the parameters are already determined in previous API calls.

```
int extra_skb;
void *ptr = NULL;
int *tag_ptr = NULL;

extra_skb = (parent_skb != NULL);
extra_skb |= sw->net_ops->port_vlan_rx(sw, dev, parent_dev, skb, forward, tag, ptr,
rx_stamp);
```

6.2.20 get_port_state

```
u8 get_port_state ( struct net_device *dev, struct net_device
**br_dev );
```

Parameters	`struct net_device *dev`	Network device instance.
	`struct net_device **br_dev`	Buffer to hold the bridge network device.
Return	`u8`	The STP state.
Description		This function retrieves the STP state of the port.

This function returns the STP state of the port associated with the network device and the pointer to the bridge network device. As the code to get these information are different in many Linux kernel version, it is necessary for the host network driver to define the function. A version for Linux 3.3 is provided.

```
static u8 get_port_state(struct net_device *dev, struct net_device **br_dev)
{
	struct net_bridge_port *p;
	u8 state;

	/* This state is not defined in kernel. */
	state = STP_STATE_SIMPLE;
	if (br_port_exists(dev)) {
```

```
            p = br_port_get_rcu(dev);
            state = p->state;

            /* Port is under bridge. */
            *br_dev = p->br->dev;
      }
      return state;
}
```

6.2.21 sw_get_priv_state

```
u8 sw_get_priv_state ( struct net_device *dev );
```

Parameters	`struct net_device *dev`	Network device instance.
Return	`u8`	STP state of the device.
Description		This function returns the STP state of the network device.

This function returns the STP state of the network device. As the private data stored in the network device are different in each host network controller, it is necessary to define this function in the host network device driver and pass it to the switch structure during switch initialization.

```
static u8 get_priv_state(struct net_device *dev)
{
      struct dev_priv *priv = netdev_priv(dev);

      return priv->state;
}
```

6.2.22 sw_set_priv_state

```
void sw_set_priv_state ( struct net_device *dev, u8 state );
```

Parameters	`struct net_device *dev`	Network device instance.
	`u8 state`	STP state of the network device.
Return		None.
Description		This procedure sets the STP state of the network device.

This procedure is used to set the STP state of the network device.

```
static void set_priv_state(struct net_device *dev, u8 state)
```

```
{
	struct dev_priv *priv = netdev_priv(dev);

	priv->state = state;
}
```

6.2.23 sw_set_multi

```
void sw_set_multi ( struct ksz_sw *sw, struct net_device *dev );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct net_device *dev`	Network device instance.
Return		None.
Description		This procedure stores the multicast addresses to be accepted.

This procedure is used to remember which multicast addresses are accepted to the host controller so that the switch can filter those addresses as promiscuous mode is used. It is called within the network device set receive mode function.

```
struct dev_priv *priv = netdev_priv(dev);
int flags = dev->flags;
int multicast = (dev->flags & IFF_ALLMULTI);

if (sw->dev_count > 1 ) {
	if ((flags & IFF_MULTICAST) && !netdev_mc_empty(dev))
		sw->net_ops->set_multi(sw, dev);
	priv->multi_list_size = 0;

	/* Do not update multi_list_size. */
	if (flags & IFF_ALLMULTI)
		flags &= ~IFF_MULTICAST;

	/* Turn on all multicast. */
	multicast |= (dev->flags & IFF_MULTICAST);
}
```

6.2.24 sw_stp_rx

```
int sw_stp_rx ( struct ksz_sw *sw, struct net_device *dev, struct
sk_buff *skb, int port, int *forward );
```

Parameters	`struct ksz_sw *sw`	Switch instance.

	`struct net_device *dev`	Network device instance.
	`struct sk_buff *skb`	Received socket buffer.
	`int port`	Receive port.
	`int *forward`	Buffer to hold the forward rule.
Return	`int`	Error code.
Description		This function determines whether to accept the received frame or not.

This function determines whether to accept the received frame or not when STP is used. If accepted the forward rule will be retrieved from the stored MAC table. With exceptions for special MAC addresses stored in the hardware MAC table, all other addresses will be remembered so that the frame can be blocked when the STP daemon forwards it to the other ports.

```
if (sw->net_ops->stp_rx(sw, dev, skb, rx_port, &forward)) {
	if (!forward) {
		if (!sw->net_ops->blocked_rx(sw, skb->data))
			printk("rxd%d=%02x:%02x:%02x:%02x:%02x:\n",
				rx_port,
				skb->data[0], skb->data[1],
				skb->data[2], skb->data[3],
				skb->data[4], skb->data[5]);
		dev_kfree_skb_irq(skb);
		return 0;
	}
}
```

6.2.25 sw_blocked_rx

```
int sw_blocked_rx ( struct ksz_sw *sw, u8 *data );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`u8 *data`	Destination MAC address.
Return	`int`	Error code.
Description		This function checks the blocked address is in the database or not.

This function checks whether the blocked address is in the database or not. It is used for debug purpose only.

6.2.26 monitor_ports

```
void monitor_ports ( struct ksz_sw *sw );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
Return		None.
Description		This procedure monitors the switch ports.

This procedure monitors all the switch ports which are under the control of STP and shuts off the port as necessary depending on its STP state. It is called by a monitor timer which is started by the `sw_open` procedure. If STP is not enabled then this procedure will not be called and the timer will not be renewed.

6.2.27 init_sw_sysfs

```
int init_sw_sysfs ( struct ksz_sw *sw, struct ksz_sw_sysfs *info,
struct device *dev );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct ksz_sw_sysfs *info`	Switch Sysfs instance.
	`struct device *dev`	Device pointer.
Return	`int`	Error code.
Description		This function setup the PTP Sysfs variables..

This function is used to setup the switch Sysfs variables so that some switch features can be accessed from user space. The `ksz_sw_sys` structure needs to be defined somewhere, as there can be more than 1 instance. The device used can be network, SPI, or I2C device.

```
err = init_sw_sysfs(sw, &sw_sysfs, &dev->dev);
```

6.2.28 exit_sw_sysfs

```
void exit_sw_sysfs ( struct ksz_sw *sw, struct ksz_sw_sysfs
*info, struct device *dev );
```

Parameters	`struct ksz_sw *sw`	Switch instance.
	`struct ksz_sw_sysfs *info`	Switch Sysfs instance.

	`struct device *dev`	Device pointer.
Return		None.
Description		This function removes the PTP Sysfs variables..

This procedure is used to remove the switch Sysfs variables when the network driver exits.

```
exit_sw_sysfs(sw, &sw_sysfs, &dev->dev);
```

6.3 PTP Functions

Some switches have 1588 PTP capability, and so a PTP driver is provided and it has its own set of API functions. The driver is included in the switch driver and so some of the PTP functions are called inside the switch driver code. Nevertheless the PTP driver still needs the host network driver to implement some code to support its operation.

The PTP driver code is included when the configuration `CONFIG_1588_PTP` is defined. If PTP function actually exists the switch's feature set will include it. This is indicated by the `PTP_HW` bit in the switch `features` variable. The `ptp_hw` structure holds all the information necessary for PTP operation.

All of the PTP register access functions are accessed through the `reg` field in the PTP structure. The standard PTP functions are accessed through the `ops` field. Only the standard functions are described here.

Samples of register access functions are:

```
ptp->reg->read
ptp->reg->write
```

Samples of hardware access functions are:

```
ptp->reg->start
ptp->reg->get_time
ptp->reg->set_time
```

Samples of standard functions are:

```
ptp->ops->init
ptp->ops->stop
```

6.3.1 ptp_init

```
void ptp_init ( struct ptp_info *ptp, u8 *mac_addr );
```

Parameters	`struct ptp_info *ptp`	PTP instance.
	`u8 *mac_addr`	The host MAC address.
Return		None.
Description		This procedure initializes the PTP system.

This procedure is used to initialize the PTP system upon network driver initialization. The host MAC address is used to set the PTP identity for debug purpose only. There are several procedures to define first if the system can support them.

```
u32 get_clk_cnt(void)
{
	/* Used with system_bus_clock to provide time in microseconds. */
	return KS_R(KS8692_TIMER1_COUNTER);
}

void test_access_time(struct ptp_info *ptp) {
	/* Find out the average read and write register delays. */
	ptp->get_delay = 100000;
	ptp->set_delay = 100000;
	if (ptp->get_delay < 10000)
		ptp->delay_ticks = 10 * HZ / 1000;
	else
		ptp->delay_ticks = 20 * HZ / 1000;
}
```

The `get_clk_cnt` function returns clock in microsecond resolution to help better determine the delay in accessing PTP registers. The `test_access_time` procedure uses the PTP read time and set time functions to find out the average delay in accessing PTP registers. Faster access allows the PTP stack to process more Sync messages in higher transmit rate. Otherwise the stack needs to drop some of the Sync messages to avoid blocking the whole PTP synchronization.

```
if (sw->features & PTP_HW) {
	struct ptp_info *ptp = &sw->ptp_hw;

	ptp->test_access_time = test_access_time;
	ptp->get_clk_cnt = get_clk_cnt;
	ptp->clk_divider = system_bus_clock;

	ptp->ops->init(ptp, mac_addr);

	if (sw->features & VLAN_PORT)
		ptp->overrides |= PTP_PORT_FORWARD;
}
```

6.3.2 ptp_exit

```
void ptp_exit ( struct ptp_info *ptp );
```

Parameters	`struct ptp_info *ptp`	PTP instance.
Return		None.
Description		This procedure initializes the PTP system.

This procedure is used to free up PTP resources when network driver exits.

```
if (sw->features & PTP_HW) {
	struct ptp_info *ptp = &sw->ptp_hw;

	ptp->ops->exit(ptp);
}
```

6.3.3 ptp_start

```
void ptp_start ( struct ptp_info *ptp, int init );
```

Parameters	`struct ptp_info *ptp`	PTP instance.
	`int init`	Indication this is an initial call.
Return		None.
Description		This procedure starts the PTP system.

This procedure starts the PTP system in preparation for synchronization. It is called subsequently to make sure the PTP configurations are correct.

```
if (sw->features & PTP_HW) {
	struct ptp_info *ptp = &sw->ptp_hw;

	ptp->reg->stop(ptp, true);
}
```

6.3.4 ptp_stop

```
void ptp_stop ( struct ptp_info *ptp );
```

Parameters	`struct ptp_info *ptp`	PTP instance.
Return		None.

Description		This procedure stops the PTP system.

This procedure is used to stop the PTP system when the network driver is stopped. It is called by the `sw_stop` function.

```
int reset = false;

if (sw->features & PTP_HW) {
	struct ptp_info *ptp = &sw->ptp_hw;

	reset = ptp->ops->stop(ptp);
}
```

6.3.5 ptp_set_identity

```
void ptp_set_identity ( struct ptp_info *ptp, u8 *addr );
```

Parameters	`struct ptp_info *ptp`	PTP instance.
	`u8 *addr`	The host MAC address.
Return		None.
Description		This procedure sets the PTP identity..

This procedure is used to update the PTP identity when the host MAC address is changed. It is used for debug purpose only.

```
if (sw->features & PTP_HW) {
	struct ptp_info *ptp = &sw->ptp_hw;

	ptp->ops->set_identity(ptp, dev->dev_addr);
}
```

6.3.6 check_ptp_msg

```
struct ptp_msg *check_ptp_msg ( u8 *data, u16 **udp_check_ptr );
```

Parameters	`u8 *data`	Received packet data.
	`u16 **udp_check_ptr`	Buffer to store the UDP checksum pointer.
Return	`struct ptp_msg *`	Returned PTP message..
Description		This function checks the packet for PTP message.

This function checks the packet for PTP message.

```
if (ptp->ops->check_msg(skb->data, NULL)) {
	/* Do something for PTP message. */
}
```

6.3.7 update_ptp_msg

```
int update_ptp_msg ( u8 *data, u32 port, u32 overrides );
```

Parameters	`u8 *data`	Transmit packet data.
	`u32 port`	Destination ports.
	`u32 overrides`	Override flags.
Return	`int`	Indication this message should be blocked..
Description		This function updates the PTP message for proper destination ports.

This function updates the PTP message by changing the destination ports so that it is only sent to open ports. If no port is open then the message will be dropped. This function is used inside the `sw_check_tx` function. It is only applicable for KSZ8463 switch.

```
void *ptr = NULL;
int (*update_msg)(u8 *data, u32 port, u32 overrides) = NULL;

if (sw->features & PTP_HW) {
	struct ptp_info *ptp = &sw->ptp_hw;

	ptr = ptp;
	update_msg = ptp->ops->update_msg;
}

if (ptp) {
	int blocked;
	u32 dst = port;
	u32 overrides = ptp->overrides;

	if (!dst && ptp->version < 1)
		dst = 3;
	if (ptp->features & PTP_PDELAY_HACK) {
		dst |= (u32) sw->tx_ports << 16;
		overrides |= PTP_UPDATE_DST_PORT;
	}
	blocked = update_msg(skb->data, dst, overrides);
	if (blocked) {
		dev_kfree_skb_irq(skb);
		return NULL;
	}
```

```
}
```

6.3.8 get_rx_tstamp

```
void get_rx_tstamp ( void *ptr, struct sk_buff *skb );
```

Parameters	`void *ptr`	PTP instance.
	`struct sk_buff *skb`	Receive socket buffer.
Return		None.
Description		This procedure returns the PTP receive timestamp using standard Linux timestamping API.

This procedure is used to return the PTP receive timestamp to the kernel using standard Linux timestamping API.

```
if (ptp_tag && (ptp->rx_en & 1))
      ptp->ops->get_rx_tstamp(ptp, skb);
```

6.3.9 get_tx_tstamp

```
void get_tx_tstamp ( struct ptp_info *ptp, struct sk_buff *skb );
```

Parameters	`struct ptp_info *ptp`	PTP instance.
	`struct sk_buff *skb`	Transmit socket buffer.
Return		None.
Description		This procedure returns the PTP transmit timestamp using standard Linux timestamping API.

This procedure is used to return the PTP transmit timestamp to the kernel using standard Linux timestamping API..

```
if (skb_shinfo(skb)->tx_flags & SKBTX_HW_TSTAMP)
      ptp->ops->get_tx_tstamp(ptp, skb);
```

6.3.10 hwtstamp_ioctl

```
int ptp_hwtstamp_ioctl ( struct ptp_info *ptp, struct ifreq
```

```
*ifr );
```

Parameters	`struct ptp_info *ptp`	PTP instance.
	`struct ifreq *ifr`	Network interface request.
Return	`int`	Error code.
Description		This function handles Linux timestamping calls.

This function is used to handle Linux timestamping calls to setup receive and transmit timestamp operation.

```
switch (cmd) {
case SIOCSHWTSTAMP:
	result = -EOPNOTSUPP;
	if (sw->features & PTP_HW)
		result = ptp->ops->hwtstamp_ioctl(ptp, ifr);
	break;
}
```

6.3.11 ptp_dev_req

```
int ptp_dev_req ( struct ptp_info *ptp, char *arg, struct
ptp_dev_info *info );
```

Parameters	`struct ptp_info *ptp`	PTP instance.
	`char *arg`	The host MAC address.
	`struct ptp_dev_info *info`	PTP device information pointer.
Return	`int`	Error code.
Description		This function handles PTP calls from applications..

This function is used to handle PTP calls from applications to execute PTP commands. A set of Micrel PTP APIs was implemented to provide PTP functionality for the applications.

```
switch (cmd) {
case SIOCDEVPRIVATE + 15:
	result = -EOPNOTSUPP;
	if (sw->features & PTP_HW)
		result = ptp->ops->dev_req(ptp, ifr->ifr_data, NULL);
	break;
}
```

6.3.12 proc_ptp_intr

```
void proc_ptp_intr ( struct ptp_info *ptp );
```

Parameters	`struct ptp_info *ptp`	PTP instance.
Return		None.
Description		This procedure processes PTP related interrupts.

This procedure is used to process interrupts related to PTP operation. It is called inside the switch interrupt handling routine.

```
if (ptp->started)
      ptp->ops->proc_intr(ptp);
```

6.3.13 ptp_drop_pkt

```
int ptp_drop_pkt ( struct ptp_info *ptp, struct sk_buff *skb, u32
vlan_id, int *tag, int *ptp_tag );
```

Parameters	`struct ptp_info *ptp`	PTP instance.
	`struct sk_buff *skb`	Receive socket buffer.
	`u32 vlan_id`	VLAN IDs defined for use with PTP.
	`int *tag`	Buffer to hold the port tag.
	`int *ptp_tag`	Buffer to hold the PTP tag.
Return		None.
Description		This function checks the receive packet for PTP message and returns indication to drop it if necessary.

This functions checks the receive packet for PTP message and may returns an indication to drop it if necessary. The tag may hold the receiving port if tail tag is not actually used. It will be reset if the VLAN IDs do not match the receiving port. The ptp_tag may be non-zero to indicate it is a PTP message.

The `get_rx_tstamp` call is called inside this function if that timestamp function is enabled. The `rx_tstamp` variable is also set to this function in case the packet is copied to forward to another network device.

The forward rule will be set to forward packet to VLAN device and main device. Note this rule will be overwritten by the `sw_stp_rx` call if STP is enabled.

```
int tag = 0;
int ptp_tag = 0;

if (sw->features & PTP_HW) {
	if (ptp->ops->drop_pkt(ptp, skb, sw->vlan_id, &tag, &ptp_tag)) {
		dev_kfree_skb_irq(skb);
		return 0;
	}
	if (ptp_tag) {
		rx_tstamp = ptp->ops->get_rx_tstamp;
		if (!forward)
			forward = FWD_VLAN_DEV | FWD_MAIN_DEV;
	}
}
```

6.3.14 get_rx_info

```
void ptp_get_rx_info ( struct ptp_info *ptp, u8 *data, u8 port,
u32 timestamp );
```

Parameters	`struct ptp_info *ptp`	PTP instance.
	`u8 *data`	Receive PTP message.
	`u8 port`	Receive port.
	`u32 timestamp`	Receive timestamp.
Return		None.
Description		This procedure keeps track of receive PTP message information.

This procedure provides the receive port and timestamp retrieved from the tail tag to the PTP driver for it to keep track of PTP messages. It is called from the `ptp_drop_pkt` function.

This is only applicable for KSZ956X switches as the procedure also provides backward compatibility for PTP stacks which use the reserved fields in the PTP header required for KSZ8463 switch..

```
*ptp_tag = sw->tag.ports & ~0x80;
ptp->ops->get_rx_info(ptp, skb->data, *ptp_tag, sw->tag.timestamp);
```

6.3.15 set_tx_info

```
void ptp_set_tx_info ( struct ptp_info *ptp, u8 *data, void
*tag );
```

Parameters	`struct ptp_info *ptp`	PTP instance.
	`u8 *data`	Transmit PTP message.
	`void *tag`	Tail tag pointer.
Return		None.
Description		This procedure updates the tail tag when sending PTP message..

This procedure updates the tail tag when sending PTP message. It is primarily used for putting in the receive timestamp of Pdelay_Req message when sending the 1-step Pdelay_Resp message. It is also used for backward compatibility for PTP stacks which use the reserved fields in the PTP header required for KSZ8463 switch. As such this is only applicable for KSZ956X switches.

```
tx_tag.ports = sw->TAIL_TAG_LOOKUP;
tx_tag.timestamp = 0;
if (ptp)
      ptp->ops->set_tx_info(ptp, skb->data, *ptp_tag, &tx_tag);
```

6.3.16 init_ptp_sysfs

```
int init_ptp_sysfs ( struct ksz_ptp_sysfs *info, struct device
*dev );
```

Parameters	`struct ksz_ptp_sysfs *info`	PTP Sysfs instance.
	`struct device *dev`	Device pointer.
Return	`int`	Error code.
Description		This function setup the PTP Sysfs variables..

This function is used to setup the PTP Sysfs variables so that some internal flags can be accessed from user space. The `ksz_ptp_sysfs` structure needs to be defined somewhere, as there can be more than 1 instance. The device used can be network, SPI, or I2C device.

```
err = init_ptp_sysfs(&ptp_sysfs, &dev->dev);
```

6.3.17 exit_ptp_sysfs

```
void exitt_ptp_sysfs ( struct ksz_ptp_sysfs *info, struct device
*dev );
```

Parameters	`struct ksz_ptp_sysfs *info`	PTP Sysfs instance.
	`struct device *dev`	Device pointer.

Return		None.
Description		This function removes the PTP Sysfs variables..

This procedure is used to remove the PTP Sysfs variables when the network driver exits.

```
exit_ptp_sysfs(&ptp_sysfs, &dev->dev);
```

7 Hardware Limitations

There are some hardware bugs in the Micrel switches related to PTP message forwarding. The software driver needs to some hacks to workaround the problems. That causes the code to be less clean and orderly.

There is a requirement that PTP peer delay messages like Pdelay_Req, Pdelay_Resp, and Pdelay_Resp_Follow_Up need to pass through a closed port. However, if the destination port field in the PTP message is set the message will go through a closed port no matter what. The driver has to watch out for those cases and modify those PTP messages accordingly or completely discard them to avoid sending a PTP Sync message through a closed port. This is only applicable for KSZ8463 switch.

Most of the Micrel switches has only 8 entries in the static MAC table. At least 4 are used for regular STP implementation. More may be required in certain situations. In that case the packets cannot be filtered at the host port completely and the network driver needs to do more work.

8 RSTP Daemon

The Linux kernel supports STP completely but not RSTP, which requires a separate user application running as a daemon. A bridge utility called `brctl` can be used to setup a bridge and link several network devices into it. For RSTP operation there is a script called `bridge-stp` that will be called by the kernel when the bridge is turned on. If the script invocation is successful the kernel assumes there is a RSTP daemon processing RSTP traffic, otherwise the kernel assumes STP is used and takes over its operation.

There are several RSTP daemons available. The one Micrel used is retrieved from https://github.com/shemminger/RSTP. This daemon has three components: `rstpd`, the daemon itself; `rstpctl`, the RSTP control utility; and `bridge-stp`, the standard Linux bridge script file to setup the daemon.

The RSTP daemon `rstpd` intercepts all STP frames received from all STP ports to determine

which one needs to shut off. It puts the ports in one of the three states—blocked, learning, or forwarding—and communicates this to the Linux kernel. The last STP state, disabled, is set if the link connection is lost.

The RSTP utility `rstpctl` is used to display the STP bridge and port information. One of the commands is `showportdetail`, which reveals whether the port is designated and forwarding, root and forwarding, or alternate and discarding.

```
rstpctl showportdetail br0
```

The order to setup the RSTP daemon is

```
rstpd
ifconfig eth0 0.0.0.0
ifconfig eth1 0.0.0.0
sleep 1
brctl addbr br0
brctl addif br0 eth0
brctl addif br0 eth1
ifconfig br0 0.0.0.0
brctl stp br0 on
rstpctl rstp br0 on
```

Generally the RSTP daemon is turned on inside the `bridge-stp` script which is invoked by the kernel when the STP bridge is turned on with the "`brctl stp br0 on`" command. However, there is a problem doing that and so that script was changed and the daemon needs to be manually turned on with the "`rstpctl rstp br0 on`" command.

Alphabetical Index