# RTL8309M_8304MB_SDK_API

# PROGRAMMING GUIDE



**V1.0.1**
**Feb 4, 2015**

**Realtek Semiconductor Corp.**
**No. 2, Innovation Road II, Hsinchu Science Park, Hsinchu 300, Taiwan**
**Tel.: +886-3-578-0211. Fax: +886-3-577-6047**
**www.realtek.com**

## COPYRIGHT



## TRADEMARKS

Realtek is a trademark of Realtek Semiconductor Corporation. Other names mentioned in this document are trademarks/registered trademarks of their respective owners.

## DISCLAIMER

Realtek provides this document “as is”, without warranty of any kind, neither expressed nor implied, including, but not limited to, the particular purpose. Realtek may make improvements and/or changes in this document or in the product described in this document at any time. This document could include technical inaccuracies or typographical errors.

## USING THIS DOCUMENT

This document is intended to be used by the software engineer when programming for Realtek RTL8309M/04MB controller chips.

Though every effort has been made to assure that this document is current and accurate, more information may have become available subsequent to the production of this guide. In that event, please contact your Realtek representative for additional information that may help in the development process.

## CONFIDENTIALITY

This document is confidential and should not be provided to a third-party without the permission of Realtek Semiconductor Corporation.

## REVISION HISTORY

| Revision | Release Date | Summary |
|---|---|---|
| 1.0.0 | 2012-10-23 | First Release |
| 1.0.1 | 2015-02-04 | Second Release. Add chip model description for RTL8309M serials. |

# Table of Contents



# Table of Tables

# Table of Figures

# 1 Overview

The RTL8309M release package contains ASIC drivers, which provides general APIs that based on user configuration to configure relative ASIC registers. The external CPU could communicate with RTL8309M through MDC/MDIO interface. But inside of the ASIC driver, it uses GPIO to emulate MDC/MDIO signal to communicate with RTL8309M. This part could be easily porting to the target platform(Please see Porting issue).

The RTL8309M serials include RTL8309N, RTL8309M, RTL8305NB and RTL8304MB. Generally RTL8309M and RTL8304MB are designed for applications with external CPU, while RTL8309N and RTL8305NB are for applications with none external CPU. So the RTL8309M release package is used for RTL8309M and RTL8304MB. The source file names within the package are prefixed with “rtl8309n” other than “rtl8309m”, but it doesn’t matter.

This document use RTL8309M to explain the usage of RTL8309M/8304MB SDK API package.

# 2 Directory Structure

RTL8309M release package is distributed with the following files:

| File name | Description |
|---|---|
| mdcmdio.c | Using two GPIO pins to emulate MDC/MDIO signals to access RTL8309M PHY registers. This file should be modified when porting the driver to other platform. |
| mdcmdio.h | MDC/MDIO API definition |
| rtk_api.c | Source code of ASIC driver high-level API |
| rtk_api.h | Constant and Structure definition for ASIC driver high-level API. |
| rtk_api_ext.h | API declaration for ASIC driver high-level API |
| rtk_error.h | Error code definition for ASIC driver high-level API |
| rtl8309n_asicdrv.c | Source code of ASIC driver low-level API |
| rtl8309n_asicdrv.h | Constant declaration for ASIC driver low-level API. |
| rtl8309n_asicdrv_ext.h | API declaration for ASIC driver low-level API. |
| rtl8309n_asictypes.h | API Port and Portmask Macro based on chip model |
| rtl8309n_types.h | Data types for ASIC driver low-level API |

# 3 ASIC Driver

## 3.1 Port Macro

The chapter 1 has explained that RTL8309M serials include RTL8309N, RTL8309M, RTL8305NB and RTL8304MB. This SDK API packaged could be used to RTL8309M and RTL8304MB. But port mapping are different for them.

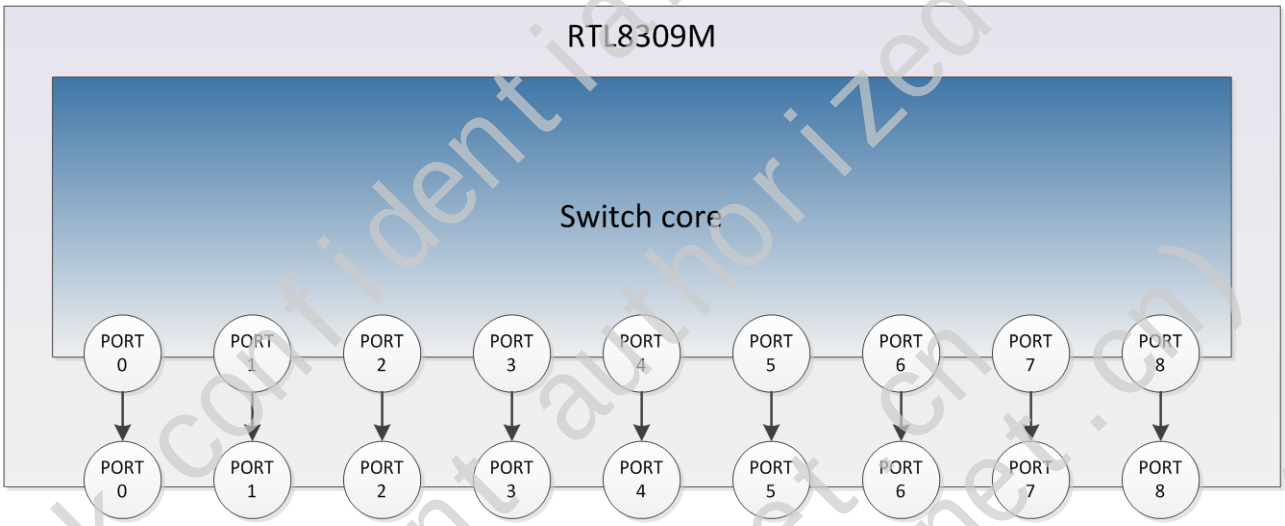



**Figure 3-1 RTL8309M and switch core port number mapping**

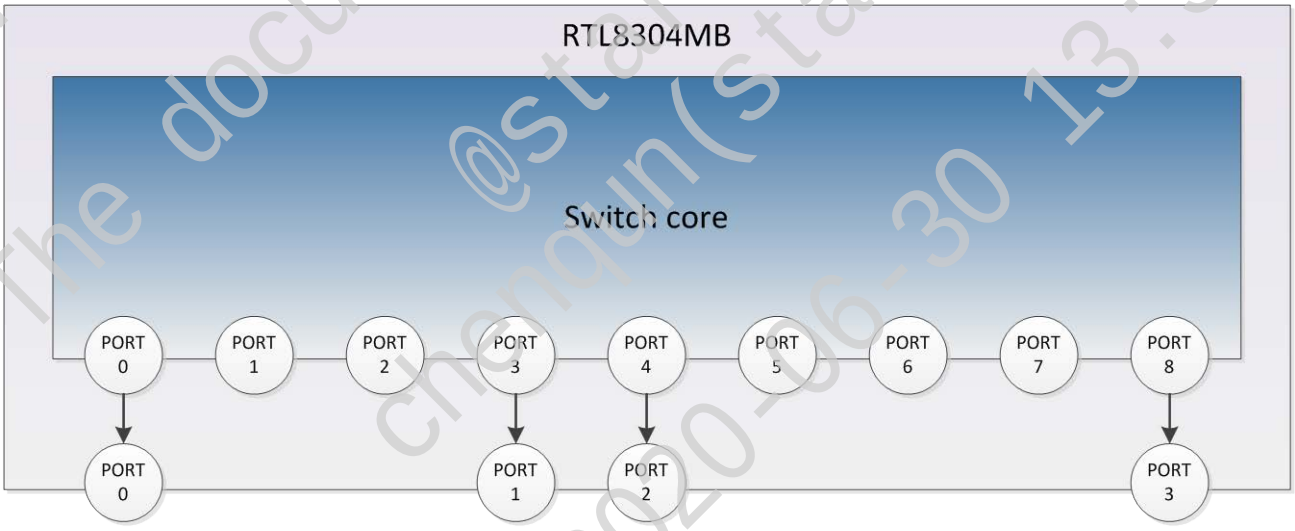



**Figure 3-2 RTL8304MB and switch core port number mapping**

| Port number internally in switch | RTL8309M port number | RTL8304MB port number |
|---|---|---|
| Port 0 | Port 0 | Port0 |
| Port 1 | Port 1 | |
| Port 2 | Port 2 | |
| Port 3 | Port 3 | Port 1 |
| Port 4 | Port 4 | Port 2 |
| Port 5 | Port 5 | |
| Port 6 | Port 6 | |
| Port 7 | Port 7 | |
| Port 8 | Port 8 | Port 3 |

**Table 3-1 Port mapping of RTL8309M and RTL8304MB**

From above figures and table the, RTL8309M port number is the same as the port number internally in the switch ASIC, but the RTL8304MB port number is different. For example, the port 1 of RTL8304MB is the port 3 in the switch ASIC.

To reduce the customer's effort, there are port and portmask macros defined in file "rtl8309n_asictypes.h". For users using RTL8309M, RTL8309M_PORT0 to RTL8309M_PORT8 could be used to call SDK API in this package. For users using RTL8304MB, RTL8304MB_PORT0 to RTL8304MB_PORT3 could be used to call SDK API in this package. User should take care of the portmask mapping. It's recommended to use port macros other than numbers for better understanding.

For example:

Example 1, the chip is RTL8304MB:

```
/* set port0, port1 as member set to VLAN 2000 */
rtk_vlan_t vid;
rtk_portmask_t mbrmsk, untagmsk;
rtk_fid_t fid;
vid = 2000;
mbrmsk.bits[0]=0x0;
```

```
SETMSKBIT(mbrmsk.bits[0], RTL8304MB_PORT0);

SETMSKBIT(mbrmsk.bits[0], RTL8304MB_PORT1);

untagmsk.bits[0]=0;

fid = 0;

rtk_vlan_set(vid, mbrmsk, untagmsk, fid);

/* set PVID of port 0/1 to 2000 and 802.1Q based default priority to 0*/

rtk_vlan_portPvid_set(RTL8304MB_PORT0, 2000, 0);

 rtk_vlan_portPvid_set(RTL8304MB_PORT1,2000, 0);
```

## 3.2 Porting issue

RTL8309M management interface is MDC/MDIO, MDC is clock and MDIO transmits data. The source code using two GPIO pins to emulate MDC/MDIO signals. Porting this driver to customer platform needs to modify files **mdcmdio.c** and **mdcmdio.h**. In these files, there are two kinds of realization, one is realized by calling GPIO API, the other is by configuring CPUGPIO register directly, any one of them is ok. If you want to write your own MDC/MDIO code, you should follow these steps:

Firstly, choose two GPIO pins, specify one as MDC, the other as MDIO;

Secondly, rewrite functions ***_smiZbit***, ***_smiReadBit***, ***_smiWriteBit***, these are local functions. ***_smiZbit*** sets two pins low to simulate HiZ state. ***_smiReadBit*** generates 1 -> 0 transition and sampled at 1 to 0 transition time, thus read one bit from MDC/MDIO interface. ***_smiWriteBit*** generates 0 -> 1 transition and put data ready during 0 to 1 whole period, it write one bit into the interface. Between MDC 1 and 0 transition, proper delay should be added, delay time will affects the interface accessing speed, the less delay, the higher speed. In IEEE 802.3 standard, it says that MDC is an aperiodic signal that has no maximum high or low times, but minimum high and low times for MDC shall be 160 ns each, and the minimum period for MDC shall be 400 ns. Please obey this limitation.

Thirdly, ***smiRead*** and ***smiWrite*** need almost no modification, smiRead reads one PHY register content, smiWrite writes value into PHY register, and both of them are global functions.

You should pay attention to another issue that is to prevent CPU from being interrupted by IRQ during smi read and write, so in ***smiRead*** and ***smiWrite,*** CPU interrupt should be disabled before smi operation and be enabled after smi operation. We use ***rtlglue_drvMutexLock()*** and ***rtlglue_drvMutexUnlock()*** which are based on RTL865X platform to realize that, if using other platform, please rewrite it.

## 3.3 Initialization

The system must be properly initialized, following figure shows a complete initialization, please pay attention to sequence of it. If you do not use the module, the module need not be initialized.

After ***rtk_switch_init, rtk_qos_init*** should be firstly called, then ***rtk_cpu_enable_set*** and ***rtk_cpu_tagPort_set***. ***rtk_qos_init*** is to configure port Tx queue numbers, it could support 4 queues at most, default only one queue. Multi-queues have different priority, higher queue could share more bandwidth of the port, thus port could be differentiated maximum four flows. ***rtk_cpu_tagPort_set*** is to specify which port is CPU port and the CPU tag insert mode.

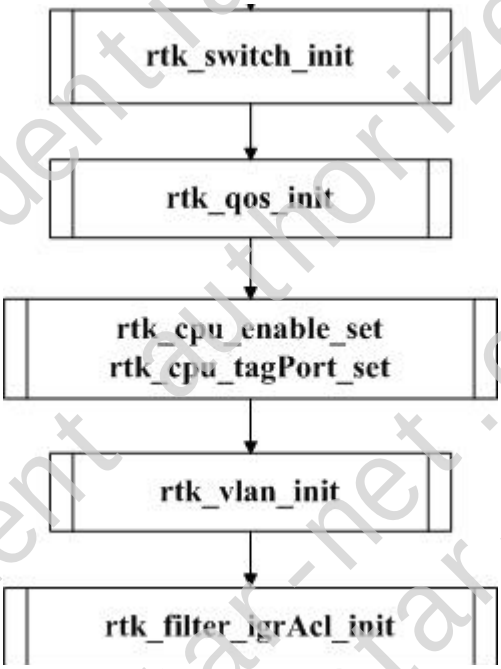



**Figure 3-3 Initialization**

# 4 RTK API for RTL8309M/8304MB

## 4.1 VLAN

RTL8309M supports 16 VLANs. After the switch resets, it is dumb mode, the VLAN function is disabled as the default configuration. User should use API ***rtk_vlan_init*** to initialize the VLAN before using VLAN. The initialization will enable VLAN function and set up a default VLAN with VID 1 that contains all ports. Moreover, each port's PVID will also be set to default VLAN.

After initialization, user can add and set VLAN 1~4094 through API ***rtk_vlan_set***, configure its member set and untag set. In order to get the information of member set, untag set and fid on specified VLAN, ***rtk_vlan_get*** could be used**.** For Port-Based VLAN, there are two APIs ***rtk_vlan_portPvid_set*** and ***rtk_vlan_portPvid_get*** provided to set and get the PVID of the ports.

User can also turn on ingress filter function through ***rtk_vlan_portIgrFilterEnable_set*** and accept frame type function through ***rtk_vlan_portAcceptFrameType_se***t.

Here are explanations and example codes for VLAN APIs. The example codes ignore the error check.

---

### *rtk_api_ret_t rtk_vlan_init(void)*

This function enables VLAN and gives the default configuration, and it should be called before using VLAN. It will clear the VLAN table and set up a default VLAN with VID 1 that contains all ports. It also sets each port's PVID to the default VLAN. After VLAN initialization, it will be set tag aware, the switch uses tagged-VID VLAN mapping for tagged frames while using Port-Based VLAN mapping for untagged frames. After invocating ***rtk_vlan_init***, user can change the default VLAN configuration through the following introduced APIs, for example ***rtk_vlan_set*** and ***rtk_vlan_portPvid_set***.

Example:

```
/* initialize VLAN */

rtk_vlan_init ();


/* all the ports are in the default VLAN 1 after VLAN initialized, modify it as follows

VLAN1

member: port0, port1, port2;

untagset : port0~port5
```

```
fid:1

VLAN2member: port3, port4, port5 ;

untagset : port0~port5

fid :2 */

rtk_portmask_t mbrmsk, untagmsk;

mbrmsk.bits[0]=0x07;

untagmsk.bits[0]=0x3F;

rtk_vlan_set(1, mbrmsk, untagmsk, 1);

mbrmsk.bits[0]=0x38;

untagmsk.bits[0]=0x3F;

rtk_vlan_set(2, mbrmsk, untagmsk, 2);


/* set PVID for each port */

rtk_vlan_portPvid_set(0, 1, 0);

rtk_vlan_portPvid_set(1, 1, 0);

rtk_vlan_portPvid_set(2, 1, 0);

rtk_vlan_portPvid_set(3, 2, 0);

rtk_vlan_portPvid_set(4, 2, 0);

rtk_vlan_portPvid_set(5, 2, 0);

rtk_vlan_portPvid_set(6, 2, 0);

rtk_vlan_portPvid_set(7, 2, 0);

rtk_vlan_portPvid_set(8, 2, 0);
```

## *rtk_api_ret_t rtk_vlan_set(rtk_vlan_t vid, rtk_portmask_t mbrmsk, rtk_portmask_t untagmsk, rtk_fid_t fid)*

```
typedef uint32    rtk_vlan_t;

typedef struct rtk_portmask_s

{

    uint32 bits[RTK_TOTAL_NUM_OF_WORD_FOR_1BIT_PORT_LIST];

} rtk_portmask_t;

typedef uint32    rtk_fid_t;
```

User could configure the member set and untag set for specified VLAN through this API. The VID should be 1~4094. The *mbrmask*'s bit N means port N. For example, *mbrmask.bits[0]* = 23=0x17=0b010111 means port 0,1,2,4 in the member set. The input parameter *untagmsk* is used to configure untag set for specified VLAN, which means if the *untagmsk* bit value is 1, the packets egress form the corresponding port will be removed VALN tag. The input parameter *fid* is for SVL/IVL usage, it should be 0~3.

Example:

```
/* set port 0,1,2,3 as member set, port 2,3 as untag set and 1 as fid to VLAN 1000 */

rtk_vlan_t vid;

rtk_portmask_t mbrmsk, untagmsk;

rtk_fid_t fid;


vid = 1000;

mbrmsk.bits[0]=0x0F;

untagmsk.bits[0]=0x0C;

fid = 1;

rtk_vlan_set(vid, mbrmsk, untagmsk, fid);
```

## *rtk_api_ret_t rtk_vlan_get(rtk_vlan_t vid, rtk_portmask_t *pMbrmsk, rtk_portmask_t *pUntagmsk, rtk_fid_t *pFid)*

*typedef uint32 rtk_vlan_t;*

*typedef struct rtk_portmask_s*

*{*

*uint32 bits[RTK_TOTAL_NUM_OF_WORD_FOR_1BIT_PORT_LIST];*

*} rtk_portmask_t;*

*typedef uint32 rtk_fid_t;*

The API can get the information of member set, untagged set, and filtering database on the specified VLAN. The information is retrieved in mbrmask, untagmsk and fid. The *mbrmask*'s bit N means port N. For example, mbrmask.bits[0] = 23=0x17=0b010111 means port 0,1,2,4 in the member set.

Example:

*/* get the member set and untagged set on VLAN 1000 */*

*rtk_vlan_t vid;*

*rtk_portmask_t mbrmsk, untagmsk;*

*rtk_fid_t fid;*

*vid = 1000;*

*rtk_vlan_get(vid, &mbrmsk, & untagmsk, &fid);*

---

## *rtk_api_ret_t rtk_vlan_destroy(rtk_vlan_t vid)*

Delete an existed VLAN, which means its VID, port member set, untagged set and fid in VALN table all will be set to 0. If the VLAN does not exist, it returns RT_ERR_VLAN_ENTRY_NOT_FOUND.

Example:

*/* delete the VLAN entry from vlan table whose VID is 2007 */*

*rtk_vlan_destroy(2007);*

---

## *rtk_api_ret_t rtk_vlan_portPvid_set(rtk_port_t port, rtk_vlan_t pvid, rtk_pri_t priority)*

Set each port's PVID and 802.1Q based default priority for the port. The specified priority will only be assigned to the untagged frame ingress from the port, and then system can map the untagged frame to the proper output queue for 802.1Q-based QoS. Just specify the priority to 0 if you don't turn on the QoS or use other QoS mechanisms instead of 802.1Q-based. Before using the API, user should call ***rtk_vlan_set*** to set the member set and untag set for the VLAN first.

Example:

```
/* set port0, port1 as member set to VLAN 2000 */

rtk_vlan_t vid;

rtk_portmask_t mbrmsk, untagmsk;

rtk_fid_t fid;


vid = 2000;

mbrmsk.bits[0]=0x03;

untagmsk.bits[0]=0;

fid = 0;

rtk_vlan_set(vid, mbrmsk, untagmsk, fid);


/* set PVID of port 0/1 to 2000 and 802.1Q based default priority to 0*/

rtk_vlan_portPvid_set(0, 2000, 0);

rtk_vlan_portPvid_set(1,2000, 0);
```

## *rtk_api_ret_t rtk_vlan_portPvid_get(rtk_port_t port, rtk_vlan_t *pPvid, rtk_pri_t *pPriority)*

Get each port's PVID and 802.1Q based default priority for the port.

Example:

```
/* get PVID and 802.1Q based default priority of port1 */

rtk_vlan_t vid;
```

```
rtk_pri_t priority;

rtk_vlan_portPvid_get(1, &vid, &priority);
```

---

## *rtk_api_ret_t rtk_vlan_portAcceptFrameType_set(rtk_port_t port, rtk_vlan_acceptFrameType_t accept_frame_type)*

Acceptable frame type could be set for each port, and priority-tagged packet (VID = 0) is treated as untagged packet.

Example:

```
/* port 0: accept tagged and un-tagged packet,*/

/* port 1: accept only tagged packet,*/

/* port 2: accept only un-tagged packet.*/

rtk_vlan_portAcceptFrameType_set (0, ACCEPT_FRAME_TYPE_ALL);

rtk_vlan_portAcceptFrameType_set (1, ACCEPT_FRAME_TYPE_TAG_ONLY);

rtk_vlan_portAcceptFrameType_set (2, ACCEPT_FRAME_TYPE_UNTAG_ONLY);
```

---

## *rtk_api_ret_t rtk_vlan_portAcceptFrameType_get(rtk_port_t port, rtk_vlan_acceptFrameType_t *pAccept_frame_type)*

Get the acceptable frame type of the port.

Example:

```
/* get acceptable frame type of port 0 */

rtk_vlan_acceptFrameType_t acceptFrameType;

rtk_vlan_portAcceptFrameType_get (0, &acceptFrameType);
```

---

## *rtk_api_ret_t rtk_vlan_portIgrFilterEnable_set (rtk_port_t port, rtk_enable_t igr_filter)*

Set VLAN ingress for each port. RTL8309M use one ingress filter configuration for whole system, not for each port, so any port you set will affect all ports ingress filter setting. While VLAN function is enabled, ASIC will decide VLAN ID for each received frame and get belonged member ports from VLAN table. If the ingress port is not included in VLAN member ports, ASIC will drop received frame if VLAN ingress function is enabled.

Example:

```
/*Enable the VLAN ingress filter function for all ports*/

rtk_vlan_portIgrFilterEnable_set(0, ENABLED);


/*Disable the VLAN ingress filter function for all ports*/

rtk_vlan_portIgrFilterEnable_set(0, DISABLED);
```

## 4.2 Lookup table

RTL8309M supports 2K-entry lookup table which supports 4-way hashing lookup, CPU could access lookup table through MDC/MDIO interface. Hashing function generates 9-bit lookup index which is the combination of MAC[0:47] and fid[0:1], for each index, there are 4-entry to save the hash collision MAC address.

MAC address stored in the lookup table does not have any constrain which means I/G-bit can be one or zero. When address with I/G-bit set to be zero, the address is unicast address, while I/G-bit one means multicast address.

Unicast address property includes static or dynamic, authorized or unauthorized. Dynamic unicast MAC address entry is auto-learned by ASIC, and after a period of time(aging time, normal 300 second), it will be aged out and removed. Static or unicast MAC address can be written into lookup table by CPU through ***rtk_l2_addr_add***. And API ***rtk_l2_addr_del can be called*** to delete it. Authorized property is also assigned by CPU, it is for IEEE 802.1x application, authorized MAC address also never age out.

Multicast address is for multicast application such as IP multicast, which could not be auto-learned but could be managed by CPU through ***rtk_l2_mcastAddr_add***. If a multicast address has been written into lookup table, packets sent to the address will be properly forwarded according to the port mask configured. Thus, they will not be flooded to all ports anymore. Multicast MAC address entry could be deleted by the API ***rtk_l2_mcastAddr_del***.

Here are explanations and example codes for lookup table API. The example codes ignore the error check.

---

### *rtk_api_ret_t rtk_l2_addr_add(rtk_mac_t *pMac, rtk_fid_t fid, rtk_l2_ucastAddr_t *pL2_data)*

```
typedef struct    rtk_mac_s
{
    uint8 octet[ETHER_ADDR_LEN];
} rtk_mac_t;
typedef uint32 rtk_fid_t;
typedef struct rtk_l2_ucastAddr_s
{
    uint8 auth;
    uint8 da_block;
    uint8 sa_block;
```

```
    uint8 isStatic;
    uint8 port;
    uint8 fid;
    uint32 age;
    rtk_mac_t mac;
}rtk_l2_ucastAddr_t;
```

Add a unicast entry to lookup table. The lowest order bit of the highest order octet (I/G-bit) of MAC address must be 0. If hashed index is full of entries that maintained by CPU, it would return error message "RT_ERR_L2_INDEXTBL_FULL". CPU should maintain a database to record the MAC addresses written to lookup table for further deletion.

Example:

```
/* write a unicast static entry with MAC address "00-12-34-56-78-AA", */
/*source port 0, IEEE 802.1x authorized, sa unblock, da unblock, fid 1 */
rtk_l2_ucastAddr_t l2_entry;
rtk_mac_t    mac;
rtk_fid_t    fid;
mac.octet[0] = 0x00;
mac.octet[1] = 0x12;
mac.octet[2] = 0x34;
mac.octet[3] = 0x56;
mac.octet[4] = 0x78;
mac.octet[5] = 0xAA;
memset(&l2_entry, 0x00, sizeof(rtk_l2_ucastAddr_t));
l2_entry.port = 0;
l2_entry.isStatic = 1;
l2_entry.auth = 1;
```

```
l2_entry.sa_block = 0;

l2_entry.da_block = 0;

l2_entry.fid = 1;

fid = 1;


rtk_l2_addr_add (&mac, fid, &l2_entry);
```

## *rtk_api_ret_t rtk_mcastAddr_add(rtk_mac_t *pMac, rtk_fid_t fid, rtk_portmask_t portmask)*

This API adds a multicast entry to lookup table. The lowest order bit of the highest order octet (I/G-bit) of MAC address must be 1.

Example:

```
/* write a multicast entry with MAC address “01-00-5E-11-22-33”, member port 0,1,2, fid 2 */

rtk_mac_t    mac;

rtk_fid_t    fid;

rtk_portmask_t    portmask;


mac.octet[0] = 0x01;

mac.octet[1] = 0x00;

mac.octet[2] = 0x5E;

mac.octet[3] = 0x11;

mac.octet[4] = 0x22;

mac.octet[5] = 0x33;

fid = 2;

portmask.bist[0] = 0x07;
```

*rtk_l2_mcastAddr_add(&mac, fid, portmask);*

---

## *rtk_api_ret_t rtk_l2_addr_del(rtk_mac_t *pMac, rtk_fid_t fid)*

Delete a unicast entry from lookup table. If the unicast mac address is not in the lookup table, it will return RT_ERR_L2_ENTRY_NOTFOUND.

Example:

*/* delete a entry with MAC address "00-12-34-56-78-AA" with fid 0*/*

*rtk_mac_t mac;*

*rtk_fid_t fid;*

*mac.octet[0] = 0x00;*

*mac.octet[1] = 0x12;*

*mac.octet[2] = 0x34;*

*mac.octet[3] = 0x56;*

*mac.octet[4] = 0x78;*

*mac.octet[5] = 0xAA;*

*fid = 0;*

*rtk_l2_addr_del(&mac, fid);*

---

## *rtk_api_ret_t rtk_l2_mcastAddr_del(rtk_mac_t *pMac, rtk_fid_t fid)*

Delete a multicast entry from lookup table. If the multicast mac address is not in the lookup table, it will return RT_ERR_L2_ENTRY_NOTFOUND.

Example:

*/* delete a entry with MAC address "01-00-5E-11-22-33"with fid 0 */*

```
rtk_mac_t    mac;
rtk_fid_t    fid;

mac.octet[0] = 0x01;
mac.octet[1] = 0x00;
mac.octet[2] = 0x5E;
mac.octet[3] = 0x11;
mac.octet[4] = 0x22;
mac.octet[5] = 0x33;
fid = 0;

rtk_l2_mcastAddr_del(&mac, fid);
```

---

### *rtk_api_ret_t rtk_l2_addr_get(rtk_mac_t *pMac, rtk_fid_t fid, rtk_l2_ucastAddr_t *pL2_data)*

```
typedef struct    rtk_mac_s
{
    uint8 octet[ETHER_ADDR_LEN];
} rtk_mac_t;
typedef uint32 rtk_fid_t;
typedef struct rtk_l2_ucastAddr_s
{
    uint8 auth;
    uint8 da_block;
    uint8 sa_block;
```

```
    uint8 isStatic;
    uint8 port;
    uint8 fid;
    uint32 age;
    rtk_mac_t mac;
}rtk_l2_ucastAddr_t;
```

Get a unicast entry from lookup table by MAC and FID. If the unicast mac address is not in the lookup table, it will return RT_ERR_L2_ENTRY_NOTFOUND.

Example:

```
/* Get a unicast entry with MAC address "00-12-34-56-78-AA"with fid 1 */
rtk_l2_ucastAddr_t    l2_entry;
rtk_mac_t    mac;
rtk_fid_t    fid;

memset(&l2_entry, 0x00, sizeof(rtk_l2_ucastAddr_t));
mac.octet[0] = 0x00;
mac.octet[1] = 0x12;
mac.octet[2] = 0x34;
mac.octet[3] = 0x56;
mac.octet[4] = 0x78;
mac.octet[5] = 0xAA;
fid = 1;

rtk_l2_addr_get(&mac, fid, &l2_entry);
```

## *rtk_api_ret_t rtk_l2_mcastAddr_get(rtk_mac_t *pMac, rtk_fid_t fid, rtk_portmask_t *pPortmask)*

Get a multicast MAC entry from lookup table by MAC and FID. If the multicast mac address is not in the lookup table, it will return RT_ERR_L2_ENTRY_NOTFOUND.

Example:

```
/* Get a multicast entry with MAC address “01-00-5E-56-78-AA” with fid 0*/

rtk_mac_t    mac;

rtk_fid_t    fid;

rtk_portmask_t    portmask;


mac.octet[0] = 0x01;

mac.octet[1] = 0x00;

mac.octet[2] = 0x5E;

mac.octet[3] = 0x56;

mac.octet[4] = 0x78;

mac.octet[5] = 0xAA;

fid = 0;


rtk_l2_mcastAddr_get(&mac, fid, &portmask);
```

## 4.3 QOS

RTL8309M QoS function provides maximum 4 queues per port for packet scheduling with queue weight and priority assignment. Priority assignment specifies the priority of each received packet according to 8 types of QOS priority sources as show in figure.4-1. Among them, CPU tag–based priority will be adopted if it is enabled. The second priority is RLDP priority. The other four priority sources are decided based on priority selection table which could be defined by user through API ***rtk_qos_priSel_set***. Priority assignment source with highest level will be selected and ASIC will use its mapped priority to locate desired queue for outputs.

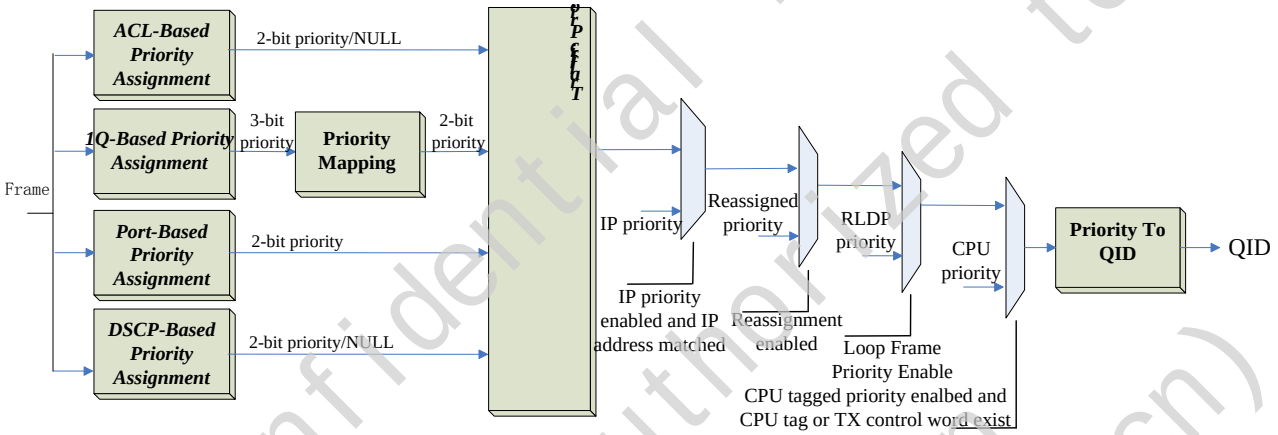



Fig. 4-1 RTL8309M Priority Assignment Diagram

If priority assignment source have the same selected priority, then ASIC will pick up the one with the largest value to be the desired queuing priority.

Each port has 4 Tx queues at most, which are queue 0~3, and packet will be classified into queues according to priority. The mapping relation between priority and QID could be specified by user. Four queues have different priority, queue 3 > queue 2 > queue 1 > queue 0, higher priority queue will occupy more bandwidth of the port when there is no queue rate limitation, it means that if all queues are full all along, queue 3 could transmit more packets than other queues, queue 0 transmits least ones. In four queues, only queue 3 and queue 2 equip leaky bucket to control queue rate. Packet scheduling block diagram is shown in following figure.

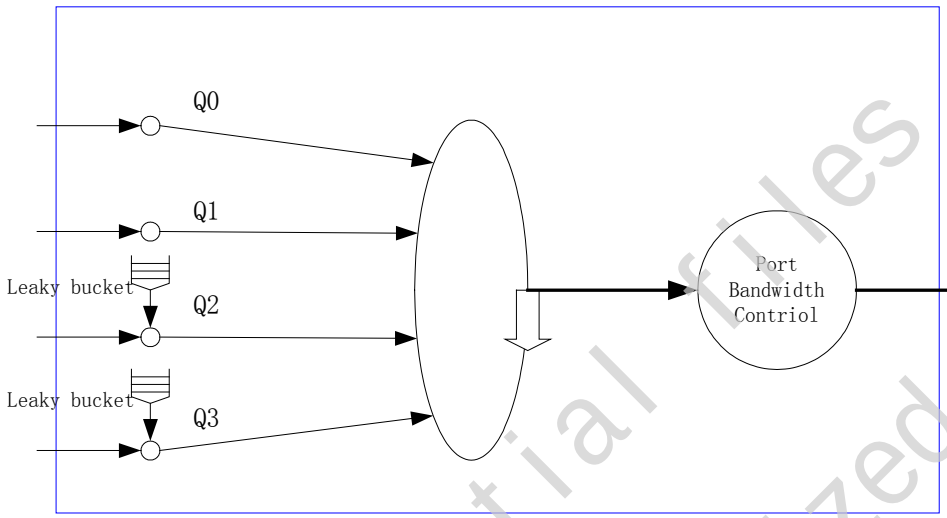



Fig. 4-2 Packet scheduling block diagram

There is a mapping table that defines the mapping between priority and queue id. After packet priority is decided, the packet will be put into table mapped queue. This table could be configured by calling API ***rtk_qos_priMap_set***. Here is default mapping table configuration when queue number is selected 4, 3, 2 and 1 after initializing QOS function by API ***rtk_qos_init***. An example is given to explain the table, in 4 queues situation, packet with priority 0 will be classified into queue 0, packet with priority 1 will be classified into queue 1, packet with priority 2 into queue 2, packet with priority 3 into queue 3.

| | Queue numbers | | | |
|---|---|---|---|---|
| Priority | 1 | 2 | 3 | 4 |
| 0(default) | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 1 | 2 |

Fig. 4-3 mapping table for priority and queue id

Besides, RTL8309M supports bandwidth control function. For each port, both input and output rate could be limited to a specified speed, rate control unit is 64Kbps, the maximum rate is 1526*64Kbps = 100Mbps. User could configure the bandwidth control by calling API ***rtk_rate_igrBandwidthCtrlRate_set*** and ***rtk_rate_egrBandwidthCtrlRate_set***. (1Kbps = 1024bps, 1Mbps = 1000,000bps)

Here are explanations and example codes for QOS API. The example codes ignore the error check

### *rtk_api_ret_t rtk_qos_init(rtk_queue_num_t queueNum)*

The *queueNum* is Tx queue number for all ports, 1 ~ 4 is permitted. This API will also assign different flow control thresholds for different queue number applications, and create a default mapping table for priority and queue id (Fig. 4-4) to locate frame to mapping queue with different priority in different queue number situation.

Example:

```
/* set 4 queues usage for each port */
rtk_qos_init(4);
```

---

### *rtk_api_ret_t rtk_qos_priSrcEnable_set(rtk_port_t port, rtk_qosPriSrc_t priSrc, rtk_enable_t enabled)*

```
typedef uint32    rtk_port_t;
typedef enum rtk_qosPriSrc_s
{
RTK_CPUTAG_PRI = 0,
RTK_IP_PRI,
RTK_DSCP_PRI,
RTK_1Q_PRI,
RTK_PORT_PRI,
RTK_QOS_PRISRC_END
}rtk_qosPriSrc_t;
typedef enum rtk_enable_e
{
    DISABLED = 0,
    ENABLED,
    RTK_ENABLE_END
} rtk_enable_t;
```

Enable priority source. RTL8309M support 8 types of priority source, and 5 types can be enabled or disabled per port. When the priority source is disabled, the priority from this priority source will be ignored during priority extraction.

*Example:*

```
/*enable port 1 dot1q based priority, enable port 3 port-based priority*/
rtk_port_t port;
rtk_qosPriSrc_t priSrc;

port = 1;
priSrc = RTK_1Q_PRI;
enabled = TRUE;
rtk_qos_priSrcEnable_set(port, priSrc, TRUE);
port = 3;
priSrc = RTK_PORT_PRI;
enabled = TRUE;
rtk_qos_priSrcEnable_set(port, priSrc, TRUE);
```

## rtk_api_ret_t rtk_qos_priSel_set(rtk_priority_select_t *pPriDec)

```
typedef struct rtk_priority_select_s
{
  uint32 acl_pri_weight;
  uint32 dot1q_pri_weight;
  uint32 port_pri_weight;
  uint32 dscp_pri_weight;
}rtk_priority_select_t;
```

As aforementioned, 8309M could recognize 8 types of priority sources at most, and a packet properly has all of them. Among them, 4 types of priority sources (ACL-based priority, DSCP-based priority, 1Q-based priority, Port-based priority) could be set weight by user through the API ***rtk_qos_priSel_set***. Each priority source could be set weight from 0 to 3, and arbitration module will decide their sequence to take. ASIC will select the highest weight assignment source's priority to decide the final priority of the packet. If two priority source have the same priority weight, then the highest priority will be choosed.

Example:

```
/* set priority arbitration weight with 802.1q > dscp > port based > ACL */

rtk_priority_select_t priDec;

priDec.dot1q_pri= 3;

priDec.dscp_pri = 2;

priDec.port_pri = 1;

priDec.acl_pri = 0;


rtk_qos_priSel_set(&priDec);
```

---

### *rtk_api_ret_t rtk_qos_dscpPriRemap_set(rtk_dscp_t dscp, rtk_pri_t int_pri)*

This API can be used to set the remapping table for translating DSCP value in IP header to internal priority. If the coming frame is not an IP frame, the priority will be NULL and ignored during priority selection process.

The value of *dscp* is selected from 0 to 63, and the range of *int_pri* is 0~3.

Example:

```
/*set DSCP value 63 mapped to priority 3, DSCP value 5 mapped to priority 0*/

rtk_qos_dscpPriRemap_set(63, 3);

rtk_qos_dscpPriRemap_set (5, 0);
```

## *rtk_api_ret_t rtk_qos_1pPriRemap_set(rtk_pri_t dot1p_pri, rtk_pri_t int_pri)*

This API can set the remapping table for translating 1Q priority mapping to internal priority that is used for queue usage and packet scheduling.

The the range of *dot1p_pri is* 0~7, and *int_pri is* 0~3.

Example:

```
/*set 1Q priority 5 mapped to internal priority 2*/

rtk_qos_1pPriRemap_set(5, 2);
```

## *rtk_api_ret_t rtk_qos_portPri_set(rtk_port_t port, rtk_pri_t int_pri)*

This API is used to configure the ingress port-based priority, and this priority should be ranged 0~3.

Example:

```
/* Set port 0~3 to priority 0 and port4~5 to priority 3 */

rtk_qos_portPri_set(0, 0);

rtk_qos_portPri_set(1, 0);

rtk_qos_portPri_set(2, 0);

rtk_qos_portPri_set(3, 0);

rtk_qos_portPri_set(4, 3);

rtk_qos_portPri_set(5, 3);
```

## *rtk_api_ret_t rtk_qos_priMap_set(rtk_queue_num_t queue_num, rtk_qos_pri2queue_t *pPri2qid)*

RTL8309M supports maximum 4 queues usage for each port and has priority mapping configuration. The API lets user define the mapping relation between internal priority and queue id. There are dedicated configurations for different queue number usage and there are different queue ids for variable queue number usage, too.

Example:

```
/*set 4 Tx queues usage and priority 0, 1 mapped to queue 0, priority 2, 3 mapped to queue 1*/

rtk_qos_pri2queue_t pri2qid;
```

```
pri2qid.pri2queue[0] = 0;

pri2qid.pri2queue[1] = 0;

pri2qid.pri2queue[2] = 1;

pri2qid.pri2queue[3] = 1;

rtk_qos_priMap_set(4, &pri2qid);
```

---

### *rtk_api_ret_t rtk_qos_1pRemarkEnable_set(rtk_port_t port, rtk_enable_t enable)*

This API is used to set per port 802.1P remarking ability for the specified port.

Example:

```
/*Enable 802.1P remarking for port 2*/RTL8309M_qos_1pRemarkEnable_set(2, ENABLED);
```

### *rtk_api_ret_t rtk_qos_1pRemark_set(rtk_pri_t int_pri, rtk_pri_t dot1p_pri)*

This API is used to configure 802.1P remarking table, which means mapping the 2-bit internal priority to 3-bit priority.

Example:

```
/*set the remarking priority of internal priority 0 to be 3*/

rtk_pri_t int_pri;

rtk_pri_t dot1p_pri ;



int_pri = 0 ;

dot1p_pri = 3 ;

rtk_qos_1pRemark_set (int_pri, dot1p_pri);
```

---

### *rtk_api_ret_t rtk_rate_igrBandwidthCtrlRate_set( rtk_port_t port, rtk_rate_t rate, rtk_enable_t ifg_include)*

This API is used to configure input bandwidth control rate for port. The *rate* unit is 64Kbps and the

range is from 64Kbps to 100Mbps. The granularity of rate is 64Kbps. The *ifg_include* parameter is used to determine rate calculation include/exclude inter-frame-gap and preamble, and its value could be following macros:

ENABLED - include inter-frame-gap and preamble;

DISABLED - exclude inter-frame-gap and preamble

Example:

```
/*Enable port1 input rate limitation and the rate is 4Mbps(62x64Kbps),*/

/*set the rate's calculation includes IFG and preamble */

rtk_rate_igrBandwidthCtrlRate_set(1, 62, ENABLED);
```

---

### *rtk_api_ret_t rtk_ rtk_rate_egrBandwidthCtrlRate_set(rtk_port_t port, rtk_rate_t rate, rtk_enable_t ifg_include))*

This API is used to configure output bandwidth control rate for port. The *rate* unit is 64Kbps and the range is from 64Kbps to 100Mbps. The granularity of rate is 64Kbps. The *ifg_include* parameter is used for rate calculation with/without inter-frame-gap and preamble.

Example:

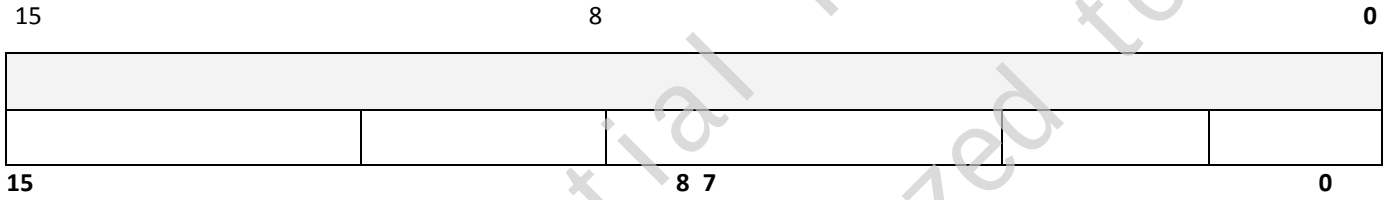
```
/*Enable port4 output rate limitation and the rate is 50Mbps(763x64Kbps),*/

/*set the rate's calculation not include IFG and preamble */

rtk_rate_egrBandwidthCtrlRate_set(4, 763, DISABLED);
```

## 4.4 CPU port

RTL8309M provides CPU port design for switch management. There is no limitation which port is CPU port and user can append any port to CPU communication as desired. ASIC will insert proprietary tag with Ethernet Length/Type 0x8899 to forward frame to CPU port as related configuration. The proprietary tag specification is as below table.




| EtherType(16-bit) [0x8899] | | | |
|---|---|---|---|
| Protocol (4-bit) [0x4] | Priority(2-bit) | Disrn(1-bit) | Tx/Rx(9-bit) |

Fig. 4-4 CPU tag format

The proprietary tag will be parsed from receiving frame from CPU port and be inserted to forward frame to CPU port only. TX/RX field of CPU tag in forwarding frame is indicated which source port that forwarding frame comes from and CPU software can use this one to apply some useful high layer applications. TX/RX field in receiving frame from CPU port is used to force layer 2 forwarding decision and ASIC will use this field as forwarding port mask with desired priority assign. Frames with CPU tag which disable learning field enabled will let the ASIC bypass source MAC learning function.

Here are explanations and example codes for CPU port API. The example codes ignore the error check

---

### *rtk_api_ret_t rtk_cpu_enable_set(rtk_enable_t enabled)*

The API can set CPU port function enable/disable, default port 8 is CPU port. User can modify CPU port setting by calling API ***rtk_cpu_tagPort_set***.

Example:

```
/*Enable CPU port function*/

rtk_cpu_enable_set (ENABLED);
```

---

### *rtk_api_ret_t rtk_cpu_tagPort_set(rtk_port_t port, rtk_enable_t enTag )*

The API can set CPU port and inserting proprietary CPU tag enable/disable for the frame that is transmitted to CPU port.

Example:

```
/*Set port 3 as CPU port and insert CPU tag */

rtk_cpu_enable_set(ENABLED);

rtk_cpu_tagPort_set(3, ENABLED);
```

---

## *rtk_api_ret_t rtk_cpu_tagPort_get(rtk_port_t *pPort, rtk_enable_t *pEnTag)*

User could use this API to retrieve current CPU port setting including current CPU port id and inserting tag enable/disable.

Example:

```
/* Get current CPU setting */

rtk_port_t port; rtk_enable_t enabled;

rtk_cpu_tagPort_get(&port, &enabled);
```

## 4.5 MIB

The RTL8309M implements sixteen 32-bit MIB Counters on each port for traffic management and diagnostic purposes. They are TX Byte Counter (64-bit), Rx Byte Counter(64-bit), Tx Packet Counter, Rx Packet Counter, Rx Drop Packet Counter, Rx CRC Packet Counter, Rx Fragment Packet Counter, Tx Broadcast Counter, Rx Broadcast Counter, Tx Multicast Counter, Rx Multicast Counter, Tx Unicast Counter, Tx Unicast Counter, Rx Symbol Counter. There is a global register for all ports to enable or disable MIB Counters. Pause frame on/off is not counted in any condition.

Here are explanations and example codes for MIB API. The example codes ignore the error check.

---

***rtk_api_ret_t rtk_mib_get(rtk_port_t port, rtk_mib_counter_t counter, rtk_mib_cntValue_t *pValue)8309M***

*typedef enum rtk_mib_counter_e*

*{*

*MIB_TXBYTECNT = 0,*

*MIB_RXBYTECNT = 2,*

*MIB_TXPKTCNT = 4,*

*MIB_RXPKTCNT,*

*MIB_RXDPCNT,*

*MIB_RXCRCCNT,*

*MIB_RXFRAGCNT,*

*MIB_TXBRDCNT,*

*MIB_RXBRDCNT,*

*MIB_TXMULTCNT,*

*MIB_RXMULTCNT,*

*MIB_TXUNICNT,*

*MIB_RXUNICNT,*

*MIB_RXSYMBLCNT = 15,*

*MIB_END*

*}rtk_mib_counter_t;*

This API is used to get the MIB counter value for the specified port and MIB type. mib counter named MIB_TXBYTECNT and MIB_RXBYTECNT are counted by unit of byte. And the counter values are 64bits long. So when these mib counter value are needed to read out, parameter pValue should be pointed to a array with 2 unsigned 32bits data elements. To read out other mib counter, the unit is packet and pValue is pointed to a unsigned 32bits value.*8309M8309M8309M8309M8309M8309M8309M8309M8309M8309M8309M8309M8309M8309M8309M8309M*

*Example:*

*/* Get RX byte counter of port 0 */*

*rtk_port_t port;*

*rtk_mib_counter_t counter;*

*uint32 cntVal[2];*

*port = 1;*

*counter = MIB_RXBYTECNT;rtk_mib_get(port, counter, &cntVal)8309M8309M;*

### *rtk_api_ret_t rtk_stat_port_reset(rtk_port_t port)*

This API is used to reset MIB counter for the specified port. The corresponding MIB counter will stop counting, clear self to 0, and then start counting again.

Example:

```
/*Reset MIB counters of port 4*/

rtk_stat_port_reset(4);
```

## 4.6 PHY

RTL8309M supports 8-port 10/100M PHYs and allows network management to configure desired capability Auto-Negotiation function allows a device to advertise modes (100F, 100H, 10F and 10H modes) of operation it possesses to a remote end of a link segment and detect corresponding operational modes. Without Auto-Negotiation function, PHY could be forced as 10/100 modes (100F, 100H, 10F and 10H modes).

Here are explanations and example codes for phy API. The example codes ignore the error check.

---

### *rtk_api_ret_t rtk_port_phyAutoNegoAbility_set(rtk_port_t port, rtk_port_phy_ability_t *pAbility)*

The structure of *rtk_port_phy_ability_t* is defined as following:

```
typedef struct rtk_port_phy_ability_s
{
    uint32      AutoNegotiation;    /*PHY register 0.12 setting for auto-negotiation process*/
    uint32      Half_10;            /*PHY register 4.5 setting for 10BASE-TX half duplex capable*/
    uint32      Full_10;            /*PHY register 4.6 setting for 10BASE-TX full duplex capable*/
    uint32      Half_100;           /*PHY register 4.7 setting for 100BASE-TX half duplex capable*/
    uint32      Full_10;            /*PHY register 4.8 setting for 100BASE-TX full duplex capable*/
    uint32      Full_1000;          /*PHY register 9.9 setting for 1000BASE-T full duplex capable*/
    uint32      FC;                 /*PHY register 4.10 setting for flow control capability*/
    uint32      AsyFC;              /*PHY register 4.11 setting for asymmetric flow controlcapability*/
} rtk_port_phy_ability_t;
```

This API enables PHY auto-negotiation and configures advertisement ability. RTL8309M switch only has 5 PHYs, so only 0~4 are permitted for input parameter *port*.

Example:

```
/*Set PHY 1 with Auto negotiation, 10F, and enable Symmetric PAUSE flow control capability*/
rtk_port_phy_ability_t    ability;
```

```
memset(&ability, 0, sizeof(ability));

ability.Full_10 = 1;

ability.FC = 1;

rtk_port_phyAutoNegoAbility_set(4, &ability);


/*set PHY 2 with Auto negotiation, 100F without flow control*/

rtk_port_phy_ability_t    ability;

memset(&ability, 0, sizeof(ability));

ability.Full_100 = 1;

rtk_port_phyAutoNegoAbility_set(2, &ability);
```

---

## *rtk_api_ret_t rtk_port_phyAutoNegoAbility_get(rtk_port_t port, rtk_port_phy_ability_t *pAbility)*

Get the capability of specified PHY. RTL8309M switch only has 5 PHYs, so only 0~4 are permitted for input parameter *port*

Example:

```
/*Get capability of PHY 1 */

rtk_port_phy_ability_t    ability;

rtk_port_phyAutoNegoAbility_get (1, &ability);
```

---

## *rtk_api_ret_t rtk_port_phyForceModeAbility_set (rtk_port_t port, rtk_port_phy_ability_t *pAbility)*

The API forces PHY as specified ability. RTL8309M switch only has 5 PHYs, so only 0~4 are permitted for input parameter *port*.

Example:

```
/*Force PHY 1 to 10HALF, and enable Symmetric PAUSE flow control capabilities*/

rtk_port_phy_ability_t    ability;
```

```
memset(&ability, 0, sizeof(ability));
ability.Half_10 = 1;
ability.FC = 1;
rtk_port_phyForceModeAbility_set (1, &ability);

/*Force PHY4 10FULL without flow control*/
rtk_port_phy_ability_t    ability;
memset(&ability, 0, sizeof(ability));
ability.Full_10 = 1;
rtk_port_phyForceModeAbility_set (4, &ability);
```

---

## *rtk_api_ret_t rtk_port_phyStatus_get(rtk_port_t port, rtk_port_linkStatus_t *pLinkStatus, rtk_port_speed_t *pSpeed, rtk_port_duplex_t *pDuplex)*

Get current PHY link status.

```
typedef enum rtk_port_linkStatus_e
{
    PORT_LINKDOWN = 0,
    PORT_LINKUP,
    PORT_LINKSTATUS_END
} rtk_port_linkStatus_t;

typedef enum rtk_port_speed_e
{
    PORT_SPEED_10M = 0,
    PORT_SPEED_100M,
```

```
    PORT_SPEED_1000M,
    PORT_SPEED_END
} rtk_port_speed_t;
typedef enum rtk_port_duplex_e
{
    PORT_HALF_DUPLEX = 0,
    PORT_FULL_DUPLEX,
    PORT_DUPLEX_END
} rtk_port_duplex_t;
```

Example:

```
/*Get link status of PHY 1 */
rtk_port_linkStatus_t linkStatus;
rtk_port_speed_t speed;
rtk_port_duplex_t duplex;

rtk_port_phyStatus_get(1, &linkStatus, &speed, &duplex);
```

8309M

## 4.7 Dot1X

RTL8309M supports IEEE 802.1X function, it includes port-based and MAC-based authentication. The following APIs are provided to configure IEEE 802.1x function.

---

### *rtk_api_ret_t rtk_dot1x_unauthPacketOper_set(rtk_port_t port, rtk_dot1x_unauth_action_t unauth_action)*

The API could set the unauthorized packet action. For RTL8309M switch, the action is for whole system, so *port* could be any value of 0~8. There are two options: dropping and trapping to CPU, dropping is the default setting.

Example:

```
/* drop unauthorized packets */

rtk_dot1x_unauthPacketOper_set(1, DOT1X_ACTION_DROP);


/* trap unauthorized packets to CPU */

rtk_dot1x_unauthPacketOper_set(0, DOT1X_ACTION_TRAP2CPU);
```

---

### *rtk_api_ret_t rtk_dot1x_portBasedEnable_set(rtk_port_t port, rtk_enable_t enabled)*

The API is used to enable or disable IEEE802.1X function by each port. When the port is enabled the function, its authentication status will determine forwarding behavior of the port.

Example:

```
/* Enable port 0 port-based IEEE 802.1x function*/

rtk_dot1x_portBasedEnable_set(0, ENABLED);


/* Disable port 1 port-based IEEE 802.1x function*/

rtk_dot1x_portBasedEnable_set(1, DISABLED);
```

---

### *rtk_api_ret_t rtk_dot1x_portBasedAuthStatus_set(rtk_port_t port,*

***rtk_dot1x_auth_status_t port_auth)***

The API is used to set port authentication status.

Example:

*/* Enable port 3, 4 port-based IEEE 802.1x function, port 3 unauthorized, port 4 authorized */*

*rtk_dot1x_portBasedEnable_set(3, ENABLED);*

*rtk_dot1x_portBasedEnable_set(4, ENABLED);*

*rtk_dot1x_portBasedAuthStatus_set(3, RTL8309M_PORT_UNAUTH);*

*rtk_dot1x_portBasedAuthStatus_set(4, RTL8309M_PORT_AUTH);*

---

***rtk_api_ret_t rtk_dot1x_portBasedDirection_set(rtk_port_t port, rtk_dot1x_direction_t port_direction)***

The API is used to set port-based operation direction, the direction could be both or in. For a packet, both direction means not only ingress port but also egress port authentication status should be checked, and when one of them is unauthorized port, the packet will be regarded as unauthorized packet. In direction means only ingress port authentication status should be checked, and a packet could be forwarded from authorized port to unauthorized port.

Example:

*/* port 3 both direction, port 4 in direction*/*

*rtk_dot1x_portBasedDirection_set (3, RTL8309M_PORT_BOTHDIR);*

*rtk_dot1x_portBasedDirection_set (4, RTL8309M_PORT_INDIR);*

---

***rtk_api_ret_t rtk_dot1x_macBasedEnable_set(rtk_port_t port, rtk_enable_t enabled)***RTL8309M supports mac-based 802.1x function. MAC address authentication status of packet will be checked, and the MAC address authentication status is stored in ASIC MAC address table. This API is used to enable/disable mac-based 802.1x function.

Example:

*/* Enable port 4 mac-based 802.1x function*/*

*rtk_dot1x_macBasedEnable_set (4, ENABLED);*

---

## *rtk_api_ret_t rtk_dot1x_macBasedDirection_set(rtk_dot1x_direction_t mac_direction)*

The API is used to set mac-based 802.1x operation direction, which is similar to port-based operation direction, including both and in direction. Both direction means both SA and DA authentication status should be checked, in direction means only SA check is required.

Example:

*/* Set mac-based 802.1x operation direction as both */*

*rtk_dot1x_macBasedDirection_set (RTL8309M_MAC_BOTHDIR);*

*/* Set mac-based 802.1x operation direction as in*/*

*rtk_dot1x_macBasedDirection_set (RTL8309M_MAC_INDIR);*

---

## *rtk_api_ret_t rtk_dot1x_macBasedAuthMac_add(rtk_port_t port, rtk_mac_t *pAuth_mac, rtk_fid_t fid)*

The API is used to set a MAC address to be authorized status, and parameter *port* is physical port the MAC address belongs to. The *fid is* the filtering database the MAC address belongs to. Before calling this API, the MAC address must exist in the address table, otherwise the API will return “the L2 entry not found”.

Example:

*rtk_port_t port;*

*rtk_mac_t auth_mac;*

*rtk_fid_t fid;*

*rtk_l2_ucastAddr_t l2_data;*

*/* Set mac-based 802.1x operation direction as in*/*

*rtk_dot1x_macBasedDirection_set (RTL8309M_MAC_INDIR);*

```
/* Enable port 5 mac-based 802.1x function*/
port = 5;
rtk_dot1x_macBasedEnable_set (port, ENABLED);

/*add authorized mac 00:12:34:56:78:9a */
memset(&l2_data, 0, sizeof(l2_data));
auth_mac.octet[0] = 0x0;
auth_mac.octet[1] = 0x12;
auth_mac.octet[2] = 0x34;
auth_mac.octet[3] = 0x56;
auth_mac.octet[4] = 0x78;
auth_mac.octet[5] = 0x9a;
fid = 0x2;
l2_data.auth = 1;
l2_data.fid = fid;
l2_data.is_static = 0;
l2_data.port = port;

rtk_l2_addr_add(&auth_mac, &l2_data);
rtk_dot1x_macBasedAuthMac_add(port, &auth_mac, fid);
```

### *rtk_api_ret_t rtk_dot1x_macBasedAuthMac_del(rtk_port_t port, rtk_mac_t *pAuth_mac, rtk_fid_t fid)*

The API is used to delete a 802.1x authenticated MAC address from port. It only changes the auth status of the MAC and won't delete it from MAC address table. Parameter *port* is physical port the MAC address belongs to.

The *fid is* the filtering database the MAC address belongs to. Before calling this API, the MAC address must exist in the address table, otherwise the API will return “the L2 entry not found”.

Example:

```
/* Suppose the mac 00:12:34:56:78:9a has been added and authorized as in the example code*/
/* of API rtk_dot1x_macBasedAuthMac_add, now user can change its auth status.*/
rtk_port_t port;
rtk_mac_t auth_mac;
rtk_fid_t fid;

port = 5;
fid = 0x2;
auth_mac.octet[0] = 0x0;
auth_mac.octet[1] = 0x12;
auth_mac.octet[2] = 0x34;
auth_mac.octet[3] = 0x56;
auth_mac.octet[4] = 0x78;
auth_mac.octet[5] = 0x9a;

rtk_dot1x_macBasedAuthMac_del(port, &auth_mac, fid)
```

## 4.8 Port mirror

RTL8309M supports one set of mirroring functions for network monitoring. The monitor port is called mirroring port, the port whose traffic is to be mirrored is called mirrored port. The mirroring port can mirror packets by DA/SA or by TX/RX port. Only one port can be set as mirroring port.

Here are explanations and example codes for port Mirror API. The example codes ignore the error check.

---

***rtk_api_ret_t rtk_mirror_portBased_set(rtk_port_t mirroring_port, rtk_portmask_t *pMirrored_rx_portmask, rtk_portmask_t *pMirrored_tx_portmask)***

Example:

```
/*set port 4 to mirror transmitting frames of port 0 and receiving frames of port 1 */

rtk_portmask_t rx_portmask;

rtk_portmask_t tx_portmask;


tx_portmask.bits[0] = 0x1;

rx_portmask.bits[0] = 0x2;

rtk_mirror_portBased_set(4, &rx_portmask, &tx_portmask);
```

---

***rtk_api_ret_t rtk_mirror_macBased_set(rtk_mac_t *macAddr, rtk_enable_t enabled)***

Example:

```
/*set port 4 to mirror packets with DA/SA is 00:12:34:56:78:9a */

rtk_mac_t mirror_mac;


mirror_mac.octet[0] = 0x0;

mirror_mac.octet[1] = 0x12;
```

```
mirror_mac.octet[2] = 0x34;

mirror_mac.octet[3] = 0x56;

mirror_mac.octet[4] = 0x78;

mirror_mac.octet[5] = 0x9a;


rtk_mirror_macBased_set(&mirror_mac, RTL8309M_ENABLED);
```

## 4.9 Port Isolation

RTL8309M supports Port Isolation function. Users can use API ***rtk_port_isolation_set*** to set a permitted forwarding port mask for each source port. The permitted forwarding port mask is configured per port. All the packets switched by RTL8309M can't be forwarded to the ports which are not in the rx port's permitted forwarding port mask.

CPU force TX (TX portmask in CPU tag) function doesn't be affected by Port Isolation. Packets which coming from CPU port and have "TX portmask in CPU tag can be forwarding to any other ports. Except this, Port Isolation is the highest priority in forwarding decision.

Here are explanations and example codes for Port Isolation API. The example codes ignore the error check.

---

### *rtk_api_ret_t rtk_port_isolation_set(rtk_port_t port, rtk_portmask_t portmask)*

This API can be used to set the permitted port mask that the specified port can transmit packets to. A port can only transmit packets to ports included in permitted forwarding *portmask*.

Example:

```
/* Set port Isolation function on Port 3 and set its permitted forwarding port mask to port 0~2*/

rtk_portmask_t portmask;

portmask.bits[0] = 0x07;

rtk_port_isolation_set(3, portmask);
```

---

### *rtk_api_ret_t rtk_port_isolation_get(rtk_port_t port, rtk_portmask_t *pPortmask)*

This API can be used to get the permitted port mask that the specified port can transmit packets to. A port can only transmit packets to ports included in permitted portmask.

Example:

```
/*Get port isolation configuration on port 5 */

rtk_portmask_t portmask;

rtk_port_isolation_get(5, &portmask);
```

## 4.10 MAC Learning Limit

RTL8309M supports source MAC learning limit function. Each port could choose to limit or un-limit the source MAC learning individually via the API ***rtk_l2_limitLearningCntEnable_set***. Port which has the MAC limitation function enabled and learned too many MACs could be configured to drop or trap the packet with new source MAC to CPU by calling the *API* ***rtk_l2_limitLearningCntAction_set. rtk_l2_learningCnt_get*** is used to retrieve the number of MACs currently learned on the specified port.

---

### *rtk_api_ret_t rtk_l2_limitLearningCnt_set(rtk_port_t port, rtk_mac_cnt_t mac_cnt)*

This API can be used to set per-port auto learning MAC limit number from 0 to 31.

Example:

*/*Enable MAC learning limit on port 2, and set MAC learning limit to 20 MAC addresses*/*

*rtk_l2_limitLearningCnt_set(2, 20);*

---

### *rtk_api_ret_t rtk_l2_limitLearningCntAction_set(rtk_port_t port, rtk_l2_limitLearnCntAction_t action)*

The API can set the action for new source MAC packet when the MAC limit of a port has been reached. The action is global, so the parameter *port* could be any value.

Example:

*/* set when the MAC limit of a port has been reached, trap the packet with the new source MAC to CPU*/*

*rtk_l2_limitLearningCntAction_set(0, LIMIT_LEARN_CNT_ACTION_TO_CPU );*

*/* set when the MAC limit of a port has been reached, drop the packet with the new source MAC */*

*rtk_l2_limitLearningCntAction_set(1, LIMIT_LEARN_CNT_ACTION_DROP);*

---

### *rtk_api_ret_t rtk_l2_learningCnt_get(rtk_port_t port, rtk_mac_cnt_t *pMac_cnt)*

The API can get per-port ASIC auto learned MAC address number.

Example:

```
/*Get current MAC learning counter on Port 2 */

rtk_mac_cnt_t mac_cnt;


rtk_l2_learningCnt_get (2, &mac_cnt);
```

## 4.11 ACL

The RTL8309M ACL holds 16 entries. When a packet is received, its source port, destination port (if a TCP or UDP packet), or EtherType code (if a non IP packet), are recorded and compared to current ACL entries. If they are matched, and if physical port and protocol are also matched, the entry is valid.

Here are explanations and example codes for ACL API. The example codes ignore the error check.

---

### *rtk_api_ret_t rtk_filter_igrAcl_init(void)*

This API can initialize ACL, that means empty the ACL table.

Example:

```
/*initialize ACL module before any configuration*/
rtk_filter_igrAcl_init ( );
```

---

### *rtk_api_ret_t rtk_filter_igrAcl_rule_add(rtk_filter_rule_t *pRule)*

The structure of *rtk_filter_rule_t* is defined as following:

```
typedef struct rtk_filter_rule_e
{
    uint32 phyport;      /*the physical port where packet received*/
    uint32 protocol;     /*Acl protocol which is in the packet*/
    uint32 data;         /*16bit ether type or TCP/UDP source port, destination port value*/
    uint32 priority;     /*Acl priority assigned to the packet*/
    uint32 action;       /*How to deal with the packet*/
} rtk_filter_rule_t;
```

This API is used to add an ACL rule into ACL table.

*(1)*

The phyport could be

0~8 - port0-port8;

RTK_ACL_ANYPORT - any port;

(2)protocol could be:ACL_PRO_ETHERTYPE (ether type), ACL_PRO_TCP (TCP), ACL_PRO_UDP (UDP), ACL_PRO_TCPUDP (TCP or UDP);

(3)priority could be 0~3;

(4)action could be :

ACL_ACT_DROP, ACL_ACT_PERMIT, ACL_ACT_TRAP2CPU, ACL_ACT_MIRROR.

Example:

```
/*Add an ACL rule, phyport: port 4, protocol: ether type 0x8800(non IP packet), */
/* priority: 0x3, action: trap to CPU */
rtk_filter_rule_t rule, rule_r;

rule.phyport = 4;
rule.protocol = ACL_PRO_ETHERTYPE;
rule.data = 0x8800;
rule.priority = 0x3;
rule.action = ACL_ACT_TRAP2CPU;
rtk_filter_igrAcl_rule_add (&rule);

/*Add an ACL rule for IP packet, phyport: port 0, TCP port: 0x5555, priority: 0x2, action: drop */
rtk_filter_rule_t rule, rule_r;

rule.phyport = 0;
rule.protocol = RTL8306_ACL_TCP;
rule.data = 0x5555;
rule.priority = 0x2;
rule.action = RTL8306_ACT_DROP;
```

```
rtk_filter_igrAcl_rule_add (&rule);
```

## *rtk_api_ret_t rtk_filter_igrAcl_rule_del(rtk_filter_rule_t *pRule)*

This API is used to delete an ACL rule from ACL table. Only phyport/protocol/data field in parameter *pRule* needs to be specified. If the ACL rule does not exist in the ACL table, it will return error.

Example:

```
/*Add an ACL rule for IP packet, phyport: port 0, TCP port: 0x5555, priority: 0x2, action: drop */

rtk_filter_rule_t rule, rule_r;

rule.phyport = RTL8306_PORT0;

rule.protocol = RTL8306_ACL_TCP;

rule.data = 0x5555;

rule.priority = 0x2;

rule.action = RTL8306_ACT_DROP;

rtk_filter_igrAcl_rule_add (&rule);


/*Delete the ACL rule above */

rtk_filter_rule_t rule, rule_r;

rule.phyport = RTL8306_PORT0;

rule.protocol = RTL8306_ACL_TCP;

rule.data = 0x5555;


rtk_filter_igrAcl_rule_del(&rule);
```

## 4.12 Storm filter

RTL8309M can effectively control four-types of storms: broadcast, ,multicast, unkown multicast, and unkown DA unicast storm.

Broadcast storm: packet which DMAC is FF-FF-FF-FF-FF-FF.

Multicast storm: Packet which address is 1 for I/G bit, including known and unknown multicast storm.

Unknown unicast storm: packet which address is 0 for I/G bit and which is lookup miss.

Unknown multicast storm: packet which address is 1 for I/G bit and which is lookup miss.

RTL8309M has 3 strom filter: broadcast, unkown unicast and multicast storm filter. Multicast storm filter can be used as multicast storm filter or Unknown multicast storm filter. When used as Multicast storm filter, it can control known multicast and unknown multicast storm. When used as unknown multicast storm filter, it can only control unknown multicast storm. These 3 types of storm filter can be per port enabled or disabled independently. User can also control the storm packet rate, burst size and set the storm filter unit, which can be byte or packet. When the packet storm rate or burst size exceed its storm filter rate or burst size, the corresponded storm filter flag will be set up until software clear it.

Here are explanations and example codes for ACL API. The example codes ignore the error check.

---

***rtk_api_ret_t rtk_storm_filterEnable_set(rtk_port_t port, rtk_rate_storm_group_t storm_type, rtk_enable_t enabled)***

```
typedef uint32    rtk_port_t;
typedef enum rtk_rate_storm_group_e
{
    STORM_GROUP_UNKNOWN_UNICAST = 0,
    STORM_GROUP_UNKNOWN_MULTICAST,
    STORM_GROUP_MULTICAST,
    STORM_GROUP_BROADCAST,
    STORM_GROUP_END
} rtk_rate_storm_group_t;
typedef enum rtk_enable_e
```

```
{
    DISABLED = 0,
    ENABLED,
    RTK_ENABLE_END
} rtk_enable_t;
```

Enable 4 types storm filter per port independently.

Example:

```
    /*enable port 1 broadcast storm filter,
      Enable port3 unkown unicast storm filter
*/
   rtk_port_t port;
   rtk_rate_storm_group_t storm_type;

   Port =1;
   storm_type = STORM_GROUP_BROADCAST;
   rtk_storm_filterEnable_set(port, storm_type, TRUE);

   Port =3;
   storm_type = STORM_GROUP_UNKNOWN_UNICAST;
   rtk_storm_filterEnable_set(port, storm_type, TRUE);
```

### *rtk_api_ret_t rtk_storm_filterAttr_set(rtk_port_t port, rtk_rate_storm_group_t storm_type, rtk_storm_attr_t *pStorm_data)*

```
typedef uint32    rtk_port_t;

typedef enum rtk_rate_storm_group_e

{

    STORM_GROUP_UNKNOWN_UNICAST = 0,

    STORM_GROUP_UNKNOWN_MULTICAST,

    STORM_GROUP_MULTICAST,

    STORM_GROUP_BROADCAST,

    STORM_GROUP_END

} rtk_rate_storm_group_t;

typedef struct rtk_storm_attr_e /*add at 2011-09-04*/

{

    uint32 unit;

    uint32 rate;

    uint32 burst;

    uint32 ifg_include;

}rtk_storm_attr_t;
```

Set storm filter rate, burst unit and whether include IPG. The storm filter unit can be based byte or packet. And so the rate unit would be pps(packet per second, storm filter unit is packet) or 1Kbps(1Kbit per second, storm filter unit is byte). The burst unit would be byte (storm filter unit is byte) or packet (storm filter unit is packet). When IPG is include, the rate or burs size will added more 20 bytes for inter frame gap and preamble when the storm filter unit is byte. IPG configuration is globally for RTL8309M ,not for per port.

Example:

```
/*set port 1 broadcast storm filter unit is packet, rate = 1024pps, burst size = 20000 packet, include IPG
set port 3 unkown unicast storm filter unit is byte, rate = 4Mbps, burst size = 20000 byte, include IPG
*/
    rtk_port_t port;
    rtk_rate_storm_group_t storm_type;
    rtk_storm_attr_t data;

    port = 1;
    storm_type = STORM_GROUP_BROADCAST;
    data.unit = 0;
    data.rate = 0x400;          //rate = 1024 * 1pps = 1024pps
    data.burst = 0x4E20; //burst = 20000 packet
    data. ifg_include = TRUE;
    rtk_storm_filterAttr_set(r port, storm_type, &data);

    port = 3;
    storm_type = STORM_GROUP_UNKNOWN_UNICAST;
    data.unit = 1;
    data.rate = 0xFA0;          //rate = 4000 * 1kbps = 4Mbps
    data.burst = 0x4e20; //burst = 20000 byte
    data. ifg_include = TRUE;
    rtk_storm_filterAttr_set(r port, storm_type, &data);
```

### *rtk_api_ret_t rtk_storm_filterStatus_set(rtk_port_t port, rtk_rate_storm_group_t storm_type, rtk_enable_t enabled)*

Clear storm filter exceed status flag. When packet storm rate or burst size exceed corresponded storm filter rate or burst size, the storm filter exceed status will be set until software clear it.

Example:

```
/*clear port 1 storm filter exceed status*/
rtk_port_t port;
rtk_rate_storm_group_t storm_type;
rtk_enable_t enabled;

port = 1;
storm_type = STORM_GROUP_BROADCAST;
enabled = TRUE;
rtk_storm_filterStatus_set(port, storm_type, enabled );
```

## 4.13 Misc

### *rtk_api_ret_t rtk_switch_init(void)*

The API can set chip registers to default configuration.

### *rtk_api_ret_t rtk_switch_maxPktLen_set(rtk_switch_maxPktLen_t len)*

The API is used to set the max packet length. For 8306E, the parameter *len* colud be set as MAXPKTLEN_1522B, MAXPKTLEN_1536B, MAXPKTLEN_1552B and MAXPKTLEN_2000B.

Example:

*/*Set max packet length to 1536Bytes */*

*rtk_switch_maxPktLen_set(MAXPKTLEN_1536B);*

### *rtk_api_ret_t rtk_trap_unknownMcastPktAction_set(rtk_port_t port, rtk_mcast_type_t type, rtk_trap_mcast_action_t mcast_action)*

The API is used to set the behavior of unknown multicast packet. When an unknown multicast packet is received, switch may trap, drop this packet. The behavior is global for all ports, so parameter *port* doesn't standard for actual physical port and can be any value from 0~5.

Example:

*rtk_port_t port;*

*rtk_mcast_type_t type;*

*rtk_trap_mcast_action_t action;*

*/*Drop unknown IPV4 multicast packet */*

*port = 0;*

*type = MCAST_IPV4;*

*action = MCAST_ACTION_DROP;*

*rtk_trap_unknownMcastPktAction_set(port, type, action);*

*/*set unknown IPV6 multicast packet to be normal forwarded */*

*port = 0;*

*type = MCAST_IPV6;*

*action = MCAST_ACTION_FORWARD;*

*rtk_trap_unknownMcastPktAction_set(port, type, action);*

---

## *rtk_api_ret_t rtk_trap_igmpCtrlPktAction_set(rtk_igmp_type_t type, rtk_trap_igmp_action_t igmp_action)*

The API is used to set both IPv4 IGMP/IPv6 MLD packet with/without PPPoE header trapping function. All the 4 kinds of IGMP/MLD function can be set separately.

The igmp packet type is as following:

- IGMP_IPV4
- IGMP_MLD
- IGMP_PPPOE_IPV4
- IGMP_PPPOE_MLD

Example:

*rtk_igmp_type_t type;*

*rtk_trap_igmp_action_t action;*

*/* Set the IPv6 IGMP packet without PPPoE header to be normal forwarded */*

*type = IGMP_MLD;*

*action = IGMP_ACTION_FORWARD;*

*rtk_trap_igmpCtrlPktAction_set (type, action);*

*/* Set the IPv4 IGMP packet with PPPoE header to be trapped to CPU*/*

*type = IGMP_PPPOE_IPV4;*

*action = IGMP_ACTION_TRAP2CPU;*

rtk_trap_igmpCtrlPktAction_set (type, action);