FSL91030(M)芯片

SDK API 调用说明文档

手册版本: V1.4

武汉飞思灵微电子技术有限公司

2022 年 12 月

目 录

1 概述

本文档对 FSL91030M 系列芯片配套的 sdk 及芯片相关原理进行了说明，SDK 总体框图如图 1-1 所示。主要提供给基于 SDK 开发的软件开发人员使用。

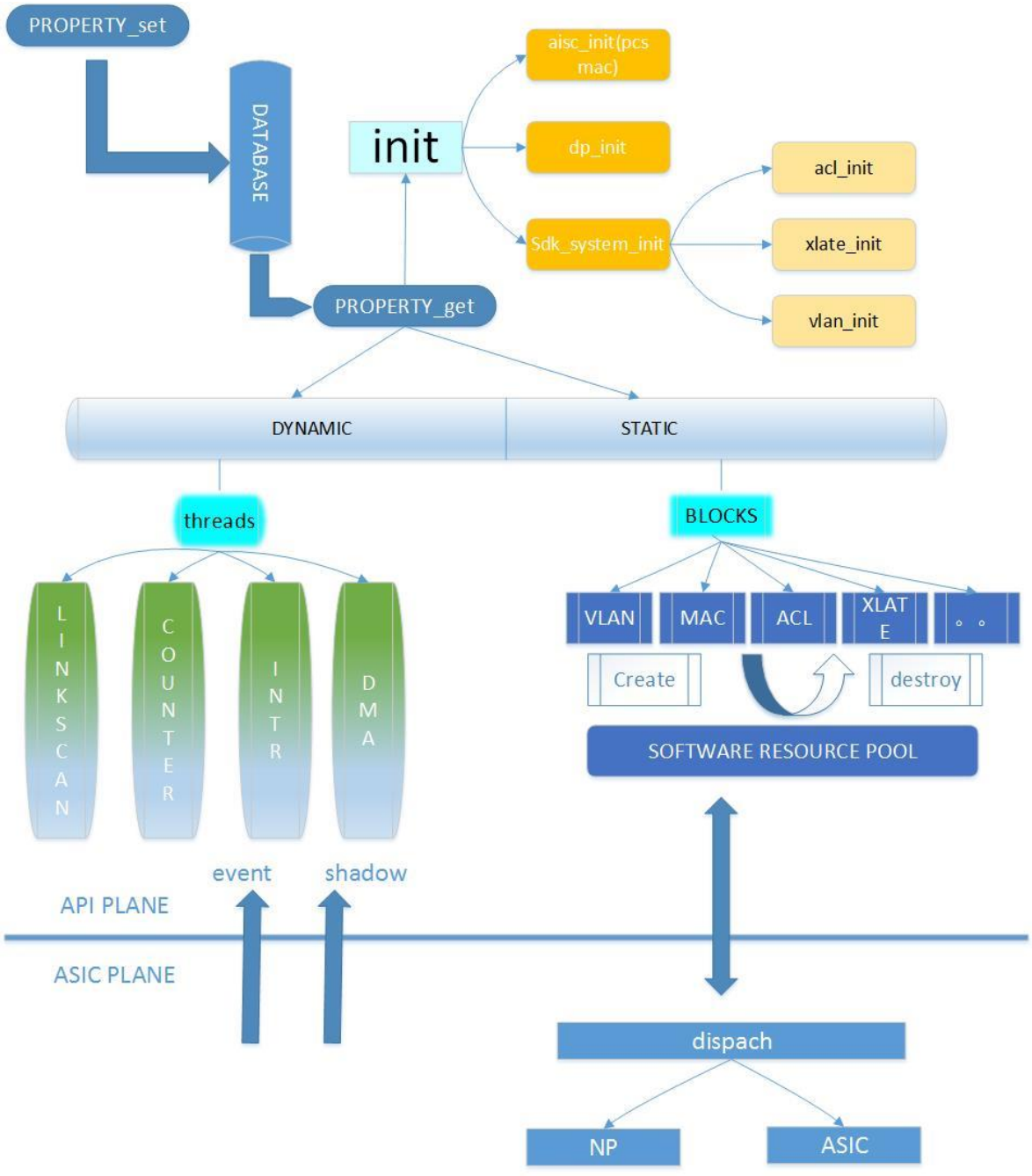
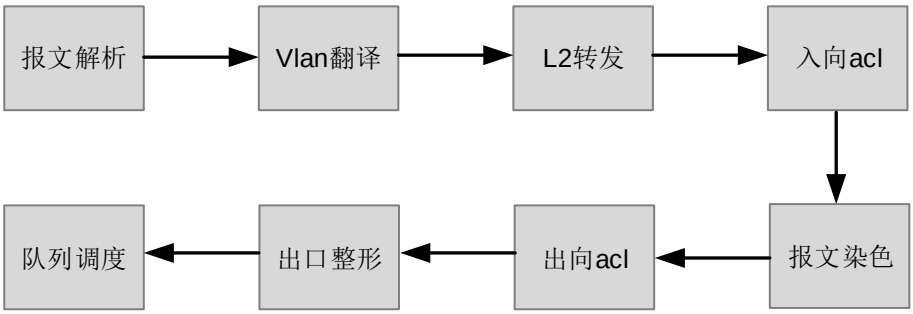


图 1-1 SDK 总体框图

FSL91030M 系列芯片可完成 2 层转发，风暴控制，acl，vlan 翻译，报文染色，优先级调度等系列功能：



每个模块，都有对应 sdk 接口为之提供相关的配置与访问。该文档结合具体配置示例，给使用者提供相关 api 的接口使用说明。同时阐述了相关的芯片原理。

2 Translate

Vlan 翻译是对报文的 vlan tag 进行处理，支持入向翻译和出向翻译，支持增加，删除，修改，复制等动作。

Vlan 翻译支持三种模式，基于 vlan，基于 mac 和基于 ip。实现过程为：匹配 vlan 翻译条目中配置的 key 信息，执行对应的 vlan 动作。

2.1 基于 vlan 的入向 vlan 翻译

入向 vlan 翻译需要使能端口的 vlan 翻译功能（支持 xlate0 和 xlate1），使能端口的 vlan 编辑功能，配置 vlan 翻译的 key 和 action；xlate0 和 xlate1 的配置方式一致，可同时配置，xlate0 优先级高。

```
int api_ingress_vlan_xlate0_action_add()

{

    int unit = 0;

    fsl_port_t port = 5;

    fsl_port_control_t type1 = fslPortControlXlateEn0;   /*xlate0 使能标识*/

    fsl_port_control_t type2 = fslPortControliVtEditEn;  /*vlan 编辑使能标识*/

    int en1 = 1;

    int en2 = 1;

    int gport = port;         /*gport 为物理端口或聚合口，此处设置为物理口*/

    int xlate = 0;

    fsl_vlan_translate_key_t key_mode = fslVlanTranslateKeyDouble; /*匹配双层 vlan, svid
和 cvid*/

    int outer_vlan = 100;

    int inner_vlan = 200;

    fsl_vlan_action_set_t action;


    /* 1.设置端口 xlate0 使能 */

    fsl_port_control_set(unit, port, type1, en1);
```

```
        /* 2.设置端口 vlan 编辑使能 */

        fsl_port_control_set(unit, port, type2, en2);

        action.new_svid = 300;

        action.new_cvid = 400;

        action.vlan_option.dtOvid = fslVlanActionReplace_dt;  /*双层 vlan 替换外层*/

        action.vlan_option.dtIvid = fslVlanActionReplace_dt;  /*双层 vlan 替换内层*/


        /* 3.设置翻译的 key 和 action */

        fsl_vlan_translate_action_add(unit, xlate, gport, key_mode, outer_vlan, inner_vlan,
&action);

            return 0;

    }
```

2.2 基于 mac 的入向 vlan 翻译

此功能需要使能端口的 vlan 翻译和 vlan 编辑功能，配置 vlan 翻译的 mac key 和 action，xlate0 和 xlate1 配置方式一致，可同时配置，xlate0 优先级高。

```
    int api_vlan_mac_action_add()

    {

        int unit = 0;

        int xlate = 1;

        int port = 4;

        fsl_vlan_translate_key_t key_mode = fslVlanTranslateKeyMac;  /*mac key*/

        fsl_mac_t mac = {0, 1, 2, 2, 3, 3};

        fsl_vlan_action_set_t action;

        int gport = port;

        int en1 = 1;
```

```
    int en2 = 1;

    fsl_port_control_t type1 = fslPortControlXlateEn0;     /*xlate0 使能标识*/

    fsl_port_control_t type2 = fslPortControliVtEditEn;    /*vlan 编辑使能标识*/


    /* 1.设置端口 xlate0 使能 */

    fsl_port_control_set(unit, port, type1, en1);


    /* 2.设置端口 vlan 编辑使能 */

    fsl_port_control_set(unit, port, type2, en2);


    /* 3.设置 vlan 翻译的 key 和 action */

    fsl_vlan_mac_action_add( unit, xlate, gport, key_mode, mac, &action);


    return 0;

}
```

2.3 mac ip 绑定

此功能需要使能端口的 vlan 翻译和 vlan 编辑功能，配置 vlan 翻译的 ip key 和 action，支持 ipv4 和 ipv6，配置 mac ip bind miss 情况下的 action(全局)，xlate0 和 xlate1 配置方式一致，可同时配置，xlate0 优先级高。

```
int api_mac_ip_bind_add()

{

    int unit = 0;

    fsl_vlan_ip_t vlan_ip;

    fsl_mac_t mac = {0, 1, 2, 3, 4, 5};

    int bypassen = 0;

    int trapen = 0;
```



```
int dropen = 1;

int port  = 10;

int en1 = 1;

int en2 = 1;

fsl_port_control_t type1 = fslPortControlXlateEn0;    /*xlate0 使能标识*/

fsl_port_control_t type2 = fslPortControliVtEditEn;   /*vlan 编辑使能标识*/

int gport = port;


/* 1.设置端口 xlate0 使能 */

fsl_port_control_set(unit, port, type1, en1);


/* 2.设置端口 vlan 编辑使能 */

fsl_port_control_set(unit, port, type2, en2);


/* 3.设置 bind miss 时的动作 */

fsl_mac_ip_bind_miss_action_set(unit, bypassen, trapen, dropen);


vlan_ip.inport = port;

vlan_ip.ip4 = 0x22446688;

vlan_ip.ip6_flag = 0;     /*配置 ipv4 时 flag 置 0, ipv6 置 1*/

vlan_ip.xlate = 0;


/* 4.设置 ip key 和 action */

fsl_vlan_ip_action_add(unit, &vlan_ip, mac, gport);
```



```
    return 0;
}
```

2.4 pdu 报文的处理

此功能需要设置 pdu 的报文头内容及其掩码，配置命中行为；其中源 mac 为全局配置，多条 pdu 配置同时命中，配置序号越小优先级越高。

```
int api_pdu_action_add()
{
    int unit = 0;
    fsl_mac_t smac = {0, 2, 3, 4, 5, 6};
    fsl_pdu_option_type_t pdu_option = PDU_OPTION_TRAP;
    int index = 5;
    fsl_pdu_config_t pdu_cfg;

    /* 1.设置 pdu 的全局源 mac */
    fsl_pdu_smac_set(unit, smac);

    /* 2.设置 pdu 识别内容和掩码 */
    pdu_cfg.ethType = 0x2345;
    pdu_cfg.mask_ethType = 1;
    fsl_pdu_config_add(unit, index, &pdu_cfg);

    /* 3.设置 pdu 动作 */
    fsl_pdu_option_set(unit, index, pdu_option);

    /*  4.设置 trap 端口为 cpu 端口 */
    fsl_traffic_trap_port_set(unit, 31); //31 口为 cpu 端口
```

```
    return 0;
}
```

2.5 基于 vlan 的出向 vlan 翻译

此功能需要使能端口 vlan 编辑功能（vlan 翻译功能的使能在 key ctl 中实现，不用单独配置），配置 vlan 翻译的 key 和 action。xlate0 和 xlate1 的配置方式一致，可同时配置，xlate0 优先级高。

```
int api_egress_vlan_xlate1_action_add()
{
    int unit = 0;
    int xlate = 1;          /*使用 xlate1*/
    int port = 8;
    int gport = port;
    fsl_vlan_translate_key_t key_mode = fslVlanTranslateKeyDouble;
    int outer_vlan = 333;
    int inner_vlan = 444;
    fsl_port_control_t type = fslPortControlEvtEditEn;  /*vlan 编辑使能标识*/
    int en = 1;
    fsl_vlan_action_set_t action;

    /* 1.设置端口 vlan 编辑使能 */
    fsl_port_control_set(unit, port, type, en);

    action.new_svid = 555;
    action.new_cvid = 555;
    action.vlan_option.dtOvid = fslVlanActionReplace_dt;  /*双层 vlan 替换外层*/
    action.vlan_option.dtIvid = fslVlanActionReplace_dt;  /*双层 vlan 替换内层*/
```



```
        /* 2.配置 vlan 翻译的 key 和 action */

        fsl_vlan_translate_egress_action_add(unit, xlate, gport, key_mode, outer_vlan,
inner_vlan, &action);



        return 0;

    }
```

2.6 端口协议 vlan 配置

该功能需要配置 protocol 的 key 和 action，protocol 的优先级高于基于 vlan、ip 和 mac 的 xlate，支持 16 配置 16 条。

```
    int api_vlan_port_protocol_action_add()

    {

        int unit = 0;

        int inIslag = 0;

        int inPort = 10;

        int ethType = 0x4567;

        fsl_vlan_action_set_t action;



        action.new_svid = 555;

        action.new_cvid = 555;

        action.vlan_option.dtOvid = fslVlanActionReplace_dt;  /*双层 vlan 替换外层*/

        action.vlan_option.dtIvid = fslVlanActionReplace_dt;  /*双层 vlan 替换内层*/



        /* 1.添加端口协议 vlan 的配置 */

        fsl_vlan_port_protocol_action_add(unit, inIslag, inPort, ethType, &action);

        return 0;

    }
```

2.7 端口 aft 滤除

```
int api_port_aft_set()

{

    int unit = 0;

    int port = 10;

    fsl_port_aft_type_t aft_type = AFT_DROP_STAG; //丢弃所有 stagged 帧


    /* 1.添加端口 aft 滤除配置 */

    fsl_port_aft_set(unit, port, aft_type);


    return 0;

}
```

3 Policing

Policing 主要是对数据流量进行控制，采用的令牌桶算法来实现流量的监管。支持基于端口和基于流两种模式。其中令牌桶算法支持 SrTCM 和 TrTCM，端口 policing 支持 35 个端口，流 policing 支持 4096 条。

3.1 端口 policing

此功能需要设置端口全局的包长计算方式，使能端口 policing 功能，设置令牌桶的刷新使能和刷新周期，最后配置限速相关参数。限速颗粒度由令牌桶刷新周期参数控制，为全局配置，刷新周期参数不能太小，最好设置在 300 以上。

令牌桶刷新设置必须在 policer 创建之前。

```
int api_macro_policing_set()

{

    int rv = 0;

    int direct = 0;

    int unit = 0;

    int port = 1;

    int enable = 1;

    fsl_policing_ctl_t pol_ctl;

    fsl_policer_config_t pol_cfg;

    fsl_policing_update_t pol_upd;


    memset(&pol_ctl, 0, sizeof(fsl_policing_ctl_t));

    pol_ctl.macroPktBytes = 0;        /*端口基于字节做 policing (1: 基于包) */

    pol_ctl.preambleLen = 20;         /*前导码和帧间隙的等效包长*/

    pol_ctl.meterGran = 0;            /*控制粒度*/


    /* 1.设置包长的计算方式 */
```

```
rv = fsl_policing_ctl_set(unit, direct, &pol_ctl);
if(rv < 0)
{
    return rv;
}
```

/* 2.端口 policing 使能 */

```
rv = fsl_macro_policing_enable_set(unit, direct, port, enable);
if(rv < 0)
{
    return rv;
}

memset(&pol_upd, 0, sizeof(fsl_policing_update_t));
pol_upd.updEn = 1;
pol_upd.updMaxIndex = 10;    /*填充令牌桶的最大地址*/
pol_upd.timer0 = 1000;     /*令牌桶的刷新周期参数，1000 个时钟周期刷新一次*/
pol_upd.timer0Num = 1;
pol_upd.timer1 = 1000;
pol_upd.timer1Num = 1;
```

/* 3.端口 policing 令牌桶刷新设置 */

```
rv = fsl_macro_policing_update_set(unit, direct, &pol_upd);
if(rv < 0)
{
    return rv;
```



```
    }


    memset(&pol_cfg, 0, sizeof(fsl_policer_config_t));
    pol_cfg.mode = fslPolicerModeTrTcm;  /*双桶双速率*/
    pol_cfg.colorSense = COLOR_BLIND;
    pol_cfg.globalCFlag = 0;
    pol_cfg.sharingMode = 0;
    pol_cfg.cir = 2000000;         /*c 桶添加速率*/
    pol_cfg.cirMax = 10000000;     /*c 桶最大添加速率*/
    pol_cfg.cbs = 4000000;         /*c 桶桶深*/
    pol_cfg.eir = 6000000;
    pol_cfg.eirMax = 8000000;
    pol_cfg.ebs = 20000000;
    pol_cfg.rChangeDrop = 1;         /*红包丢弃*/


    /* 4.创建端口的 policer */
    rv = fsl_macro_policer_create(unit, direct, port, &pol_cfg);


    return rv;
}
```

3.2 流 policing

此功能需要设置流的全局包长计算方式，设置令牌桶的刷新使能和刷新周期，最后配置限速相关参数。限速颗粒度由令牌桶刷新周期参数控制，为全局配置，刷新周期参数不能太小，最好设置在 300 以上。其中 pol_id 可设置成-1，表示 policing id 由系统分配，0-4095 为指定 id，一般情况下 flow policing id 设置为 vlan id。

令牌桶刷新设置必须在 policer 创建之前。


```
int api_flow_policing_set()
{
    int direct = 0;
    int unit = 0;
    int port = 1;
    int enable = 1;
    int pol_id = 10;              /*policng id 指定为 10*/
    fsl_policing_ctl_t pol_ctl;
    fsl_policer_config_t pol_cfg;
    fsl_policing_update_t pol_upd;

    memset(&pol_ctl, 0, sizeof(fsl_policing_ctl_t));
    pol_ctl.flowPktBytes = 0;         /*flow 模式，基于字节做 policing*/
    pol_ctl.preambleLen = 20;

    /* 1.设置包长计算方式（流）*/
    fsl_policing_ctl_set(unit, direct, &pol_ctl);

    memset(&pol_upd, 0, sizeof(fsl_policing_update_t));
    pol_upd.updEn = 1;              /*令牌桶刷新使能标识*/
    pol_upd.updMaxIndex = 10;
    pol_upd.timer0 = 1000;
    pol_upd.timer0Num = 1;
    pol_upd.timer1 = 1000;
    pol_upd.timer1Num = 1;
```

```
    /* 2.流 policing 令牌桶刷新设置 */
    fsl_flow_policing_update_set(unit, direct, &pol_upd);

    memset(&pol_cfg, 0, sizeof(fsl_policer_config_t));
    pol_cfg.mode = fslPolicerModeTrTcm;
    pol_cfg.colorSense = COLOR_BLIND;
    pol_cfg.globalCFlag = 0;
    pol_cfg.sharingMode = 0;
    pol_cfg.cir = 2000000;          /*c 桶添加速率*/
    pol_cfg.cirMax = 10000000;      /*c 桶最大添加速率*/
    pol_cfg.cbs = 4000000;          /*c 桶桶深*/
    pol_cfg.eir = 6000000;
    pol_cfg.eirMax = 8000000;
    pol_cfg.ebs = 20000000;
    pol_cfg.rChangeDrop = 1;

    /* 3.创建流的 policer */
    fsl_flow_policer_create(unit, direct, pol_id, &pol_cfg);

    return 0;
}
```


4 Map

不同的报文使用不同的 QoS 优先级，例如 vlan 报文使用 802.1p，IP 报文使用 dscp，MPLS 报文使用 exp。为了保证不同报文的服务质量，在报文进入设备时，需要将报文携带的 QoS 优先级映射到设备的内部服务等级（调度优先级）和丢弃优先级（颜色），在设备内部根据内部服务等级进行拥塞管理，根据报文颜色进行拥塞避免，在报文出设备时，需要将内部服务等级和颜色映射成 QoS 优先级，并重标记到报文中，以便后续设备根据 QoS 优先级提供相应的服务质量，这就是报文的优先级和重标记过程。

4.1 vlan 优先级到内部优先级的映射

此功能需要设置 vlan 优先级映射的模板，需要映射的 vlan 优先级，映射成的内部优先级和颜色。

```
int api_vlanpri_map_set()
{
    int unit = 0;
    int qos_pro_index = 1;              /*qos_pro_index=-1 时表示系统分配*/
    int pkt_pri = 5;                    /*报文优先级*/
    int cfi = 0;
    int internal_pri = 3;               /*内部优先级*/
    fsl_color_t color = fslColorGreen;
    fsl_qos_profile_t qos_profile;

    memset(&qos_profile, 0, sizeof(fsl_qos_profile_t));
    qos_profile.useDefault = 0;      /*0:其他，1：使用默认优先级*/
    qos_profile.useL2Info = 1;       /*0：优先使用 L3 头信息，1：使用 L2 头信息*/
    qos_profile.trustCtag = 0;      /*0：优先使用 stag，1：使用 ctag*/

    /* 1.创建模板 */
    fsl_qos_profile_create(unit, qos_pro_index, &qos_profile);
```

```
    /* 2.设置映射 */
    fsl_vlan_priority_map_set(unit, qos_pro_index, pkt_pri, cfi, internal_pri, color);

    return 0;
}
```

4.2 dscp 到内部优先级的映射

此功能需要设置 dscp 优先级映射的模板，需要映射的 dscp 值，映射成的内部优先级和颜色。

```
int api_dscp_map_set()
{
    int unit = 0;
    int qos_pro_index = 2;      /*指定使用的模板序号*/
    int dscp = 55;
    int cfi = 0;
    int internal_pri = 2;
    fsl_color_t color = fslColorYellow;
    fsl_qos_profile_t qos_profile;

    memset(&qos_profile, 0, sizeof(fsl_qos_profile_t));
    qos_profile.useDefault = 0;
    qos_profile.useL2Info = 0;      /*0：优先使用 L3 头信息，1：使用 L2 头信息*/

    /* 1.创建模板 */
    fsl_qos_profile_create(unit, qos_pro_index, &qos_profile);

    /* 2.设置映射 */
```

```
    fsl_dscp_map_set(unit, qos_pro_index, dscp, internal_pri, color);

    return 0;

}
```

4.3 内部优先级重标记 vlan pri

此功能需要使能端口的重标记功能，创建重标记模板，设置需要重标记的内部优先级、颜色和最终标记的结果 vlan pri 值。

```
int api_vlanpri_remark_set()

{

    int unit = 0;

    int port = 3;

    int rmk_en = 1;                /*重标记使能*/

    int rmkPriPtr = -1;            /*系统自动分配重标记模板序号*/

    int internal_pri  = 4;

    fsl_color_t color = fslColorGreen;

    int cos = 6;

    int cfi = 1;

    fsl_rmk_info_t rmk_info;


    /* 1.使能端口的重标记功能 */

    fsl_pri_remark_enable_set(unit, port, rmk_en);


    memset(&rmk_info, 0, sizeof(fsl_rmk_info_t));

    rmk_info.ccosRmkEn = 1;

    rmk_info.scosRmkEn = 1;


    /* 2.创建重标记模板 */
```

```
    fsl_remark_profile_create(unit, rmkPriPtr, &rmk_info);


    /* 3.设置重标记映射 */
    fsl_vlanpri_unmap_set(unit, rmkPriPtr, internal_pri, color, cos, cfi);


    return 0;
}
```

4.4 内部优先级重标记 dscp

此功能需要使能端口的重标记功能，创建重标记模板，设置需要重标记的内部优先级、颜色和最终标记的结果 dscp 值。

```
int api_dscp_remark_set()
{
    int unit = 0;
    int port = 3;
    int rmk_en = 1;
    int rmkPriPtr = -1;
    int internal_pri  = 4;
    fsl_color_t color = fslColorYellow;
    int dscp = 44;
    fsl_rmk_info_t rmk_info;


    /* 1.使能端口的重标记功能 */
    fsl_pri_remark_enable_set(unit, port, rmk_en);


    memset(&rmk_info, 0, sizeof(fsl_rmk_info_t));
    rmk_info.brgChgTos = 1;             /*重标记修改 tos*/
```

```
    rmk_info.onlyChgDscp = 1;           /*只修改 dscp，和上面的修改 tos 合在一起使用*/


    /* 2.创建重标记模板 */
    fsl_remark_profile_create(unit, rmkPriPtr, &rmk_info);


    /* 3.设置重标记映射 */
    fsl_dscp_unmap_set(unit, rmkPriPtr, internal_pri, color, dscp);


    return 0;
}
```

5 Storm control

风暴控制是为了防止局域网中广播、组播、单播包的洪泛使网络丧失性能，对网络流量进行监管的一种数据流量管理功能，通过设置流量监管门限来实现控制。

此功能支持三种模式，系统，端口和转发 id，以下 api 以端口为例。

5.1 风暴控制的全局配置

设置风暴控制的全局配置，主要是设置控制颗粒和帧间隙前导码的等效包长，使能端口的风暴控制，设置令牌刷新周期，最后设置限速值及桶尺寸。

刷新周期设置必须在限速设置之前执行。

```
int api_storm_control_set()

{

    int unit = 0;

    fsl_storm_control_mode_t mode = STORM_CONTROL_PORT;

    int gport = 8;                /*支持物理端口，聚合口，保护端口，0-46*/

    fsl_forward_type_t fwd_type = FORWARD_TYPE_MULTICASTCAST; /*转发类型，支持单播组播和广播*/

    int enable = 1;

    uint8_t meter_gran = 0;

    uint8_t preamble_len = 20;         /*帧间隙和前导码等效包长*/

    fsl_storm_policing_type_t pol_type = STORM_POLICING_BYTE;

    uint32_t limit = 1000000;

    uint32_t burst_size = 2000000;

    fsl_storm_ctl_global_t global_ctl;


    /* 1.设置风暴控制全局配置 */

    fsl_storm_control_global_set(unit, meter_gran, preamble_len);
```



```
    /* 2.使能端口风暴控制功能 */
    fsl_storm_control_enable_set(unit, mode, gport, fwd_type, enable);

    memset(&global_ctl, 0, sizeof(fsl_storm_ctl_global_t));
    global_ctl.delayInterval = 1000;
    global_ctl.maxUpdIdx = 10;
    global_ctl.updEn = 1;

    /* 3.设置令牌桶更新配置 */
    fsl_storm_control_update_set(unit, mode, &global_ctl);

    /* 4.设置限速 */
    fsl_storm_control_set(unit, mode, gport, fwd_type, pol_type, limit, burst_size);

    return 0;
}
```

6 Forwarding

描述

该功能块主要分为三个部分，转发、学习、老化，全部都是围绕着 mac 地址展开。FSL91030M 芯片中维护着两张 mac 表，分为 mac key 表和 mac action 行为表（以下统称为 mac 表），它们一一对应。mac key 表中主要存放 mac 地址、vlanid、valid 指示等，其对应的行为表则决定 mac 地址一系列行为属性，包括端口号、黑白名单、站点移位优先级、静态指示等。

转发：该部分主要决定报文的转发行为，在转发数据时通过查找报文的目的 mac，看其是否在 mac 地址表中，来决定报文是走单播、组播、洪泛等逻辑。查找方法是通过 keytype(默认是 0) +Mac + vlanid 来组成 key，利用 key 来 hash 出一个 index 来查找 mac 表，如果查找命中则根据 mac 行为表中的端口属性来决定报文的出端口。例如：mac 表中存在 mac 为 0x11,vlanid=4095,port=1,现在在 vlan 为 4095，端口 port=2 中进来一个报文，其目的 mac 为 0x11，那么报文转发命中将只会从端口 port =1 中转发，从而减少广播洪泛。

学习：开启 mac 地址学习功能，对于报文的源 mac 地址，会将其学习到 mac 表中包括对应的行为属性会学习到行为表中，学习到的都是动态 mac 地址。mac 地址学习也分为学习命中，命中后会对 mac 行为表更新。

老化：开启老化后，会在一定时间周期对 mac 表进行老化，时间周期可以配置，而开启快速老化会立刻对 mac 表进行老化。

目的

1. 通过调用接口手动添加\删除 mac 地址表：手动添加的 mac 地址表为静态表，并可以添加 mac 行为表的行为属性。
2. Mac 地址表老化时间可配置
3. 基于接口/VLAN 关闭学习 MAC 能力
4. 基于接口/VLAN/全局进行 MAC 地址数限制

6.1 添加一条 mac 地址

```
int api_l2_addr_add()

{

    sal_mac_addr_t mac = {0x00, 0x33, 0x44, 0x55, 0x66, 0x77};

    fsl_vlan_t vid = 4095;
```

```
        /*mac 行为表标志位，默认为静态地址（下面标志分别表示，静态地址、trunk 地址、黑名单、白名单）*/

        uint32_t  flags  =  FSL_L2_STATIC  |  FSL_L2_TRUNK_MEMBER  |  FSL_L2_DISCARD_SRC  |
FSL_L2_WHITE_LIST ;

        fsl_l2_addr_t l2addr;



        /*  mac 地址结构体初始化*/

        fsl_l2_addr_t_init(&l2addr, mac, vid);

        l2addr.port = 20;

        l2addr.flags = flags;

        /*trunk id*/

        l2addr.tgid = 3;



        /*添加一条 mac 地址*/

        fsl_l2_addr_add( 0, &l2addr);

        return 0;

    }
```

6.2 删除一条 mac 地址

```
    int api_l2_addr_delete()

    {

        sal_mac_addr_t mac = {0x00, 0x33, 0x44, 0x55, 0x66, 0x77};

        fsl_vlan_t vid = 4095;

        fslral_mem_t  mem;

        uint32_t key_index;

        fsl_l2_addr_t l2addr;

        int rv;

        /*先判断该 mac 地址是否在 mac 地址表中，存在才删除，并返回该 mac 地址的表类型和索引*/
```



```
    rv = fsl_l2_addr_get( 0, mac,  vid,  &l2addr,  &mem, &key_index);
    if(FSLRAL_E_NONE == rv)
    {
        fsl_l2_addr_delete( 0,  mac,  vid,  mem, key_index);
    }
    return 0;
}
```

6.3 基于 vlan 来删除 mac 地址

```
int api_l2_addr_delete_by_vlan()
{
    fsl_vlan_t vid = 4095;
    int rv;
    uint32_t flags;
    /*删除该 vlan 域下的所有 mac 地址*/
    rv = fsl_l2_addr_delete_by_vlan(0, vid, flags);

    return 0;
}
```

6.4 基于端口来删除 mac 地址

```
int api_l2_addr_delete_by_port()
{
    int rv;
    uint32_t flags;
    fsl_port_t port = 20;
    fsl_module_t mod;
    /*删除该端口下的所有 mac 地址*/
```

```
    rv = fsl_l2_addr_delete_by_port( 0,  mod,  port,  flags);
    return 0;
}
```

6.5 创建一个组播组 id

创建一个组播组 id，并添加组播成员。

```
int api_mcast_create()
{
    int group_id = 4094;
    fsl_pbmp_t pbmp;
    int rv;
    memset(&pbmp, 0, sizeof(fsl_pbmp_t));

    /*设置组播组成员*/
    pbmp.pbits[0] = 0xf;

    /*初始化全局变量*/
    fsl_mcast_init(0);
    rv = fsl_mcast_create(0,  group_id,  pbmp);
    return 0;
}
```

6.6 删除一个组播组 id

删除一个组播组 id，并清空组播成员。

```
int api_mcast_delete()
{
    int group_id = 4094;
    int rv;
```



```
    rv = fsl_mcast_delete( 0,  group_id);


    return 0;
}
```

6.7 添加一条组播类型的 mac 地址

```
int api_mcast_addr_add()
{
    sal_mac_addr_t mac_address = {0x00, 0x11, 0x44, 0x55, 0x66, 0x77};
    fsl_vlan_t vid = 4095;
    fsl_mcast_addr_t mcaddr;
    int group_id = 4094;


    /*初始化组播地址结构体*/
    fsl_mcast_addr_t_init(&mcaddr, mac_address, vid);
    mcaddr.group = group_id;
    fsl_mcast_addr_add( 0,  &mcaddr);


    return 0;
}
```

6.8 删除组播组端口成员

```
int api_mcast_bitmap_del()
{
    int group_id = 4094;
    fsl_pbmp_t pbmp;
    memset(&pbmp, 0, sizeof(fsl_pbmp_t));
```

```
    /*删除组播 id 为 4094 中的 0 1 成员*/
    pbmp.pbits[0] = 0x3;
    fsl_mcast_bitmap_del( 0,  group_id,  pbmp);

    return 0;
}
```

6.9 删除一条组播类型 mac 地址

```
int api_mcast_addr_remove()
{
    sal_mac_addr_t mac_address = {0x00, 0x11, 0x44, 0x55, 0x66, 0x77};
    fsl_vlan_t vid = 4095;
    int rv;
    /*删除一条组播类型 mac 地址*/
    rv = fsl_mcast_addr_remove( 0,  mac_address,  vid);
    return 0;
}
```

7 Vlan

介绍

VLAN（Virtual Local Area Network）即虚拟局域网，是将一个物理的 LAN 在逻辑上划分成多个广播域（多个 VLAN）的通信技术。VLAN 内的主机间可以直接通信，而 VLAN 间不能直接互通，从而将广播报文限制在一个 VLAN 内。由于 VLAN 间不能直接互访，因此提高了网络安全性，FSL91030M 芯片支持基于 VLAN 的转发，同时定义了每个 vlan 的属性，包括 VLAN 的端口成员、TRUNK 成员、黑白名单模式、优先级等，通过创建一条 vlan 域来确定该 vlan 内的一系列行为。

目的

1. 添加\删除 vlan 域

2. 为 vlan 域添加属性等

例如：创建一条 vlan 域 vlan=4095,为其添加端口成员 0 1 2 3 pbmp=0xf，那么带有 vlan=4095 从端口 1 中进来的普通报文将会从 0 2 3 口转出。

7.1 创建一条 vlan 域

```
int api_vlan_create()

{

    fsl_vlan_t vid = 4095;

    /*创建一条 vlan 域*/

    fsl_vlan_create( 0, vid);



    return 0;

}
```

7.2 删除一条 vlan 域

```
int api_vlan_destroy()

{

    fsl_vlan_t vid = 4095;

    /* 删除一条 vlan 域*/

    fsl_vlan_destroy( 0,  vid);
```

```
    return 0;
}
```

7.3 删除所有 vlan 域

```
int api_vlan_destroy_all()
{
    int rv;
    /*删除所有 vlan 域*/
    rv = fsl_vlan_destroy_all(0);
}
```

7.4 基于 vlan 来添加端口成员

```
int api_vlan_port_add()
{
    fsl_vlan_t vid = 4095;
    fsl_pbmp_t pbmp,  ubmp,  lbmp;
    memset(&pbmp, 0, sizeof(fsl_pbmp_t));
    memset(&ubmp, 0, sizeof(fsl_pbmp_t));
    memset(&lbmp, 0, sizeof(fsl_pbmp_t));
    pbmp.pbits[0] = 0xff;
    ubmp.pbits[0] = 0xf;
    lbmp.pbits[0] = 0x3;

    fsl_vlan_port_add( 0,  vid,  pbmp,  ubmp,  lbmp);

    return 0;
}
```

7.5 基于 vlan 来删除端口成员


```
int api_vlan_port_remove()
{
    fsl_vlan_t vid = 4095;
    fsl_pbmp_t pbmp,  lbmp;
    memset(&pbmp, 0, sizeof(fsl_pbmp_t));
    memset(&lbmp, 0, sizeof(fsl_pbmp_t));

    /*待移除成员*/
    pbmp.pbits[0] = 0xf;
    lbmp.pbits[0] = 0x1;
    fsl_vlan_port_remove( 0,  vid,  pbmp,  lbmp);
    return 0;
}
```

7.6 设置入方向端口 stp 状态

```
int api_ingress_stp_state_set()
{
    /*gport 指示为一个本地端口*/
    fsl_port_t gport = 0x000008;
    fsl_vlan_t vid = 4095;
    int stp_state = FSL_STP_BLOCKING;

    /*开启入方向 stp 和 erps check 使能*/
    fsl_ingress_stp_erps_enable_set(0, gport, fslPortControlStpChEn, 1);
    fsl_ingress_stp_erps_enable_set(0, gport, fslPortControlerpsLkpEn, 1);

    /*设置入方向 stpid*/
```

```
    fsl_vlan_control_set(0, fslVlanStpId, vid, 63);

    /*设置端口 stp 状态*/
    fsl_ingress_port_stp_set(0,  gport,  vid,  stp_state);

    /*gport 指示为一个 trunk id=1 端口成员假设为 0 1 2*/
    gport = 0xc000001;

    /* 开启入方向 stp 和 erps check 使能*/
    fsl_ingress_stp_erps_enable_set(0, gport, fslPortControlStpChEn, 1);
    fsl_ingress_stp_erps_enable_set(0, gport, fslPortControlerpsLkpEn, 1);

    /*设置入方向 stpid*/
    fsl_vlan_control_set(0, fslVlanStpId, vid, 63);

    /* 设置端口 stp 状态*/
    stp_state = FSL_STP_LEARNING;
    fsl_ingress_port_stp_set(0,  gport,  vid,  stp_state);

    return 0;
}
```

7.7 获取入方向端口 stp 状态

```
int api_ingress_stp_state_get()
{
    fsl_port_t gport = 0x000008;
```

```
    fsl_vlan_t vid = 4095;
    int stp_state ;
    /*获取端口 stp 状态*/
    fsl_ingress_port_stp_get(0,  gport,  vid,  &stp_state);
    gport = 0xc000001;
    fsl_ingress_port_stp_get(0,  gport,  vid,  &stp_state);
    return 0;
}
```

7.8 设置出方向端口 stp 状态

```
int api_egress_stp_state_set()
{
    fsl_port_t gport = 0x000004;
    fsl_vlan_t vid = 4095;
    int stp_state = FSL_STP_LEARNING;

    /* 开启出方向 stp 和 erps check 使能*/
    fsl_egress_stp_erps_enable_set(0, gport, fslPortControlOutStpChkEn, 1);
    fsl_egress_stp_erps_enable_set(0, gport, fslPortControlEerpsLkEn, 1);

    /*设置出方向 stpid*/
    fsl_vlan_control_set(0, fslVlanOutStpId, vid, 127);

    /* 设置端口 stp 状态*/
    fsl_egress_port_stp_set(0,  gport,  vid,  stp_state);
    gport = 0xc000002;
    stp_state = FSL_STP_BLOCKING;
```

```
    fsl_egress_stp_erps_enable_set(0, gport, fslPortControlOutStpChkEn, 1);
    fsl_egress_stp_erps_enable_set(0, gport, fslPortControlEerpsLkEn, 1);
    fsl_egress_port_stp_set(0,  gport,  vid,  stp_state);

    return 0;
}
```

7.9 获取出方向端口 stp 状态

```
int api_egress_stp_state_get()
{
    fsl_port_t gport = 0x000004;
    fsl_vlan_t vid = 4095;
    int stp_state;

    /*获取端口 stp 状态*/
    fsl_egress_port_stp_get(0,  gport,  vid,  &stp_state);
    gport = 0xc000002;
    fsl_egress_port_stp_get(0,  gport,  vid,  &stp_state);

    return 0;
}
```

7.10 添加端口 tpid 类型

```
int api_port_tpid_add()
{
    fsl_port_t port = 1;
    uint16_t tpid = 0x88a8;
```



```
    /*tpid 初始化，放在初始化代码中*/
    fsl_port_tpid_init(0);
    fsl_port_tpid_add( 0,  port, tpid);
    return 0;
}
```

7.11 删除端口 tpid 类型

```
int api_port_tpid_delete()
{
    fsl_port_t port = 1;
    uint16_t tpid = 0x88a8;

    /* tpid 初始化，放在初始化代码中*/
    fsl_port_tpid_init(0);
    fsl_port_tpid_delete( 0, port, tpid);

    return 0;
}
```

7.12 设置端口 tpid 类型

```
int api_port_tpid_set()
{
    fsl_port_t port = 1;
    uint16_t tpid = 0x88a8;

    // tpid 初始化，放在初始化代码中
    fsl_port_tpid_init(0);
    fsl_port_tpid_set( 0, port, tpid);
```

```
    return 0;
}
```

7.13 设置入方向端口 erps 环网保护

示例中，为 vlan = 4095 设置环网保护的入方向成员端口 pbmp = 0x3,在 vlan = 4095 的 vlan 域中，端口 gport=4 开启了 erps 保护使能，其不在保护成员端口中，所以从 4 口进来的报文会被丢弃。

```
int api_vlan_port_ingress_erps_set()
{
    fsl_vlan_t vid = 4095;
    fsl_pbmp_t pbmp;
    fsl_pbmp_t lbmp;

    /*gport 指示为一个普通本地端口*/
   gport=0x0000004
    memset(&pbmp, 0, sizeof(fsl_pbmp_t));
    memset(&lbmp, 0, sizeof(fsl_pbmp_t));

    /* 开启入方向 erps check 使能*/
    fsl_ingress_stp_erps_enable_set(0, gport, fslPortControlerpsLkpEn, 1);

    /*入方向只允许 0 1 口通过*/
    pbmp.pbits[0] = 0x3;
    fsl_vlan_control_set( 0,  fslVlanErpsId,  vid,  2);
    fsl_vlan_port_ingress_erps_set( 0,  vid,  pbmp,  lbmp);

    return 0;
}
```

7.14 设置出方向端口 erps 环网保护。

在示例中，为 vlan = 4095 设置环网保护的出方向成员端口 pbmp = 0xc,在 vlan = 4095 的 vlan 域中，为出端口 gport=2 开启了 erps 保护使能，其在保护成员端口中，所以从 2 口出去的报文会被正常转发。

```
int api_vlan_port_egress_erps_set()

{

    fsl_vlan_t vid = 4095;

    fsl_pbmp_t pbmp;

    fsl_pbmp_t lbmp;

   gport= 0x0000004;

    memset(&pbmp, 0, sizeof(fsl_pbmp_t));

    memset(&lbmp, 0, sizeof(fsl_pbmp_t));


    /*开启出发向 eprs check 使能*/

    fsl_egress_stp_erps_enable_set(0, gport, fslPortControlEerpsLkEn, 1);


    /*出方向只允许 2 3 口通过*/

    pbmp.pbits[0] = 0xc;

    fsl_vlan_control_set( 0, fslVlanOutErpsId,  vid,  3);

    fsl_vlan_port_egress_erps_set( 0,  vid,  pbmp,  lbmp);

    return 0;

}
```

8 Field Processer

字段处理器可以对报文的各种协议字段进行报文分类，也可以对用户自定义的协议字段进行分类。根据这些分类，可以采取不同的操作，例如丢弃报文、发送数据包到中央处理器、修改 VLAN 字段等。

每个 FIeld Processor 条目被分配到特定的组中。每个组都能够基于一组特定的限定属性来匹配数据包。每个组中可以包含多个条目，每个条目支持多个协议字段的匹配，同时指定多个不同的操作行为。

1. 创建限定属性的字段组

- fsl_field_group_create_mode_id(int unit,fsl_field_qset_t qset,int pri
- fsl_field_group_mode_t mode,uint16_t entry_num
- fsl_field_key_tp_t key_tp, fsl_field_group_t group)

Qset 指定该组用于的 stage，仅支持:

- fslFieldQualifyStageIngress(入口 acl)
- fslFieldQualifyStageEgress(出口 acl)
- fslFieldQualifyStageLooku(入口 vlan 变换)
- fslFieldQualifyStageLookupEgress(出口 vlan 变换)。

Key_Tp 指定该组支持的 key 类型。不同 key 类型支持的匹配协议字段不同。

Mode 指定该组模式。可以配置该组支持的匹配协议字段个数。

表 8-1 入口 Acl 模块中不同 key_tp 和 mode 支持的协议字段

Mode / Key tp	ModeSingle	ModeDouble	ModeTriple	ModeQuad
KEY_TP_0	InPorts、SrcMac、DstMac...	EtherType 、Stag、Ctag...	—	—
KEY_TP_1	InPorts、SrcIp、DstIp、L4SrcPort 、L4DstPort...	TcpControl...	—	—
KEY_TP_2	InPorts、DstIp6...	SrcIp6、Tos...	Stag、Ctag、DstMac、EtherType...	L4SrcPort、L4DstPort、TcpControl...
KEY_TP_3	InPorts、SrcMac、	L4SrcPort、L4DstPort、	DstIp6、Ttl...	SrcIp6、TcpControl...

	DstMac、EtherType ...	SrcIp、DstIp...		

2. 创建 entry 条目

fsl_field_entry_create_id(int unit,fsl_field_group_t group,fsl_field_entry_t entry)

3. 配置匹配协议字段

例如匹配报文 DstIp 字段：

fsl_field_qualify_DstIp(int unit, fsl_field_entry_t entry, fsl_ip_t data, fsl_ip_t mask)

4. 指定匹配后的操作行为

fsl_field_action_add(int unit, fsl_field_entry_t entry, fsl_field_action_t action,

uint32_t param0, uint32_t param1)

5. 写入硬件表项

fsl_field_entry_create_id(int unit,fsl_field_group_t group,fsl_field_entry_t entry)

8.1 调用示例

以入口 Acl 为例，输入端口 1 入的流匹配 DstIp 字段，匹配后的行为为重定向到端口 2 输出。

```
int api_fp_test(int unit)
{
    int                     rv = 0;
    fsl_field_group_t       group_id;
    fsl_field_entry_t       entry_id;
    fsl_field_qualify_t         stage;
    fsl_field_key_tp_t      key_tp;
    fsl_field_group_mode_t  mode;
    uint16_t                entry_num;
    fsl_field_qset_t        qset;
    fsl_port_t              inport;
    fsl_port_t              redirect_port;
```

```
fsl_port_control_t      type;
fsl_ip_t                dip;
fsl_ip_t                dip_mask;
group_id    = 1;
stage       = fslFieldQualifyStageIngress;
key_tp      = _FSL_FIELD_KEY_TP_1;
mode        = fslFieldGroupModeDouble;
entry_num   = 100;
inport      = 1;
redirect_port   = 2;
type        = fslPortControliAcl0LkpVld;
```

/*基于入端口开启入 Acl 查询功能*/

```
rv = fsl_port_control_set(unit, inport, type, 1);
    If (rv) {
        return rv;
    }
```

/*fp 初始化，需最先被调用一次*/

```
rv = fsl_field_init(unit);
if (rv) {
    return rv;
}

sal_memset(&qset, 0, sizeof(fsl_field_qset_t));
FSL_FIELD_QSET_ADD(qset, stage);
```

/*创建 group,需指定 stage,key_tp,mode,entry 条目数。group ID 全局唯一*/

```
        rv = fsl_field_group_create_mode_id(unit, qset, 0, mode, entry_num, key_tp,
group_id);
        if (rv) {
            return rv;
        }

        entry_id = 1;
```

/*创建 entry,创建 entry 之前需保证 group 已经创建,添加匹配项。entry ID 全局唯一*/

```
        rv = fsl_field_entry_create_id(unit, group_id, entry_id);
        if (rv) {
            return rv;
        }
        dip          =  0xc0a80001;
        dip_mask     =  0xffffff00;
```

/*该 entry 匹配目的 ipv4 地址*/

```
        rv = fsl_field_qualify_DstIp(unit, entry_id, dip, dip_mask);
        if (rv) {
            return rv;
        }
```

/*指定匹配行为为重定向*/

```
        rv = fsl_field_action_add(unit, entry_id, fslFieldActionRedirectPort, redirect_port,
0);
        if (rv) {
            return rv;
```



```
        }
        /*写硬件表项*/
        rv = fsl_field_entry_install(unit, entry_id);
        if (rv) {
            return rv;
        }
        return rv;
    }
```

9 Trunking(Link Aggregation)

Trunk(也称为端口捆绑、链路聚合)是一种将许多以太网口捆绑在一起形成 trunk 的方法。Trunk 被认为是一条逻辑链路，当交换机需要高带宽时非常有用。Trunk 有许多有价值的特性，如同源同宿、链路冗余(如果 trunk 口出现故障，则将出现故障的端口从 trunk 中移除)。

9.1 调用示例

创建一个 trunk 组，将多个端口加入到该 trunk 组，并开启 failover 保护功能。当 trunk 组中端口成员 link 状态由 up 变为 down 时，则将该端口流量负载均衡到 trunk 组*中其他端口输出；当端口 link 状态由 down 恢复到 up 时，流量从该端口恢复输出。

```
int api_trunk_test(int unit)

{

    int                 rv = 0;

    fsl_trunk_t             tid;

    int                 psc;

    fsl_trunk_add_info_t    add_info;


    /*trunk 初始化，需首先被调用*/

    rv = fsl_trunk_init(unit);

    if (rv) {

        return rv;

    }


    /*创建 trunk 组，最多支持 8 个 trunk 组，trunk ID 范围为 0~7*/

    tid = 1;

    rv = fsl_trunk_create_id(unit, tid);

    if (rv) {

        return rv;

    }
```

/*设置 hash key，例如本例以 sip+dip 计数 hash 值*/

```
psc = FSL_TRUNK_PSC_SRCIP | FSL_TRUNK_PSC_DSTIP;
rv = fsl_trunk_psc_set(unit, tid, psc);
if (rv) {
    return rv;
}
memset(&add_info, 0, sizeof(fsl_trunk_add_info_t));
add_info.num_ports  = 4;
add_info.psc            = psc;
add_info.alg            = HASH_CRC8_DEFAULT;
add_info.tp[0]          = 1;
add_info.tp[0]          = 5;
add_info.tp[0]          = 6;
add_info.tp[0]          = 10;
```

/*添加 trunk 组的端口成员，同时指定 hash key 和 hash 算法类型。同一个端口只能属于一个 trunk 组*/

```
rv = fsl_trunk_set(unit, tid, &add_info);
if (rv) {
    return rv;
}
```

/*开启 trunk 组保护功能。当 trunk 组中端口成员 link 状态由 up 变为 down 时，则将该端口流量负载均衡到 trunk 组中其他端口输出；当端口 link 状态由 down 恢复到 up 时，流量从该端口恢复输出。*/

```
rv = fsl_trunk_failover_set(unit, tid, 1);
if (rv) {
    return rv;
```

```
        }

        return rv;

    }
```

10 TM

TM 模块支持队列的调度和管理，支持基于端口和队列的流量整形，支持基于队列的 wred 功能。

10.1 端口整形

对端口的流量使用令牌桶进行整形，单桶模式。

```
int api_port_traffic_shape_set()
{
    int unit = 0;
    int port = 1;  /* outport */
    fsl_shape_mode_t mode = BYTE_MODE;
    int fill_rate = 10000;  /* 限速 10000kb */
    uint16_t burst_size = 100000;
    uint8_t quantum = 2;   /* 桶深颗粒度 */
    fsl_port_traffic_shape_set(unit, port, mode, fill_rate, burst_size, quantum);
    return 0;
}
```

10.2 队列整形

对某一个端口的特定队列使用令牌桶进行整形，双桶模式。

```
int api_port_traffic_shape_set()
{
    int unit = 0;
    int port = 2;  /* outport */
    uint8_t queue = 3;
    fsl_shape_mode_t mode = BYTE_MODE;
    int c_fill_rate = 10000;  /*保证带宽*/
    int p_fill_rate = 20000;
```



```
    uint16_t c_burst_size = 100000;
    uint16_t p_burst_size = 300000;
    uint8_t quantum = 2;    /* 桶深颗粒度 */
    fsl_queue_traffic_shape_set(unit, port, queue, mode, c_fill_rate,
                        p_fill_rate, c_burst_size, p_burst_size, quantum)
    return 0;
}
```

10.3 队列调度

对进入队列的包设置调度模式，支持 SP、WRR 和 DWRR 调度。

```
int api_queue_traffic_schedule_set()
{
    int unit = 0;
    int port = 4;
    fsl_queue_schedule_t que_sch_cfg;
    memset(&que_sch_cfg, 0, sizeof(fsl_queue_schedule_t));
    que_sch_cfg.wrr_pri = 3; /*WRR/DWRR 调度优先级，若与 SP 调度优先级相同则 SP 调度优先*/
    que_sch_cfg.sch_mode = 1; /*DWRR 模式*/
    que_sch_cfg.sch_bmp = 0x1c; /*采用 DWRR 模式的队列为 2/3/4*/
    que_sch_cfg.pri2_wrr_weight = 2; /*DWRR 调度权重*/
    que_sch_cfg.pri3_wrr_weight = 4;
    que_sch_cfg.pri4_wrr_weight = 6;
    fsl_queue_traffic_schedule_set(unit, port, &que_sch_cfg);
    return 0;
}
```


11 Pkt DMA

Pkt DMA 用于内置 cpu 的收发包。在使用 Pkt DMA 前，需要先安装网络驱动 xy1000_net.ko。该 ko 主要有两个功能：作为内置 cpu 的网络驱动，将调试网口的包上送协议栈；将非调试网口上满足过滤规则的包，根据过滤规则上送应用层或剥掉 vlan 后上送协议栈。

驱动安装后，ifconfig -a 可以查看到有 eth0 和 vir0 两个网络设备。两个网络设备共用同一个收发包硬件，其中 eth0 对应调试网口，vir0 对应其他的带内面板口。

其中，调试网口为交换芯片面板口中的一个，安装驱动后，可以通过 cat 命令查看，也可以通过 echo 命令修改。

```
cat /sys/module/xy1000_net/parameters/net_port

echo 0 > /sys/module/xy1000_net/parameters/net_port
```

非调试网口的过滤器，以及应用层收发包，都提供有 API 函数。

11.1 过滤器配置

过滤器是用于过滤从非调试网口送到 cpu 内核的报文（从调试网口送到 cpu 的报文无需过滤，直接上送协议栈），过滤器可以设置报文的过滤规则，以及行为。

规律规则是将报文中的相应字段，与过滤器指定的掩码进行与操作，如果跟过滤器指定的字段相等，则按照过滤器指定的行为处理报文。

过滤器的行为有三种：不作处理（KCOM_DEST_T_NULL）、上送协议栈（KCOM_DEST_T_NETIF）、上送应用层（KCOM_DEST_T_API）。

过滤器按照优先级进行匹配报文，优先级越小，等级越高，越先进行报文过滤。

```
#define KCOM_FILTER_MAX  256

#define KCOM_DEST_T_NULL        0

#define KCOM_DEST_T_NETIF       1

#define KCOM_DEST_T_API         2

typedef uint8_t sal_mac_addr_t[6];


void main()

{
```

```
        sal_mac_addr_t mac,mac_mask;
        uint16_t type,type_mask;
        uint16_t dest;
        uint8_t ids_num,ids[KCOM_FILTER_MAX];
        kcom_filter_t filter;

        /*创建过滤器"dmac_filter"，报文偏移为 0，长度为 6 (即报文的 dmac 字段)：
        * mac =0x11:0x22:0x33:0x44:0x55:0x66
        * mask=0xff:0xff:0xff:0xff:0xff:0xff
        * 行为：报文偏移为 0，长度为 6 的字段，和 mask 做与运算，如果等于 mac & mask，则上送应用层
        */
        mac={0x11,0x22,0x33,0x44,0x55,0x66};
        memset(mac_mask,0xff,6);
        dest=KCOM_DEST_T_API;
        if(fsl_rx_filter_create(0,6,mac,mac_mask,5,"dmac_filter",strlen("dmac_filter"),de
st))
        {
            printf("create filter error\n");
        }

        /*创建过滤器"type_filter"，报文偏移为 12，长度为 2 (即报文的 type 字段)：
        * type = 0x0806
        * mask = 0xffff
        * 行为：报文偏移为 12，长度为 2 的字段，和 mask 做与运算，如果等于 type & mask，则上送协议栈
        */
        type= 0x0806;
        type_mask =0xffff
```



```
        type = FSLOS_ENDIAN_CPU_TO_BIG_U16(type);

        dest=KCOM_DEST_T_NETIF;

        if(fsl_rx_filter_create(12,2,&type,&type_mask,4,"type_filter",strlen("type_filter
"),dest))

        {

            printf("create filter error\n");

        }



        /*查询已创建的过滤器的 id 及数量*/

        if(fsl_rx_filter_list(ids,&ids_num))

        {

            printf("get filter list error\n");

        }

        printf("get filter list[%d]: ",ids_num);

        for(i=0;i<ids_num;i++)

        {

            printf("%d ",ids[i]);

        }

        printf("\n");



        /*根据过滤器 id 查看过滤器，并打印过滤器信息*/

        if(fsl_rx_filter_get(ids[0],&filter))

        {

            printf("get filter error\n");

            return;

        }

        printf("filter.desc: %s\n",filter.desc);
```

```
printf("filter.id: %d\n",filter.id);
printf("filter.priority: %d\n",filter.priority);
printf("filter.dest_type: ",filter.dest_type);
switch(filter.dest_type)
{
    case KCOM_DEST_T_NULL:
        printf("KCOM_DEST_T_NULL\n");
        break;
    case KCOM_DEST_T_NETIF:
        printf("KCOM_DEST_T_NETIF\n");
        break;
    case KCOM_DEST_T_API:
        printf("KCOM_DEST_T_API\n");
        break;
    default:
        break;
}
printf("filter.pkt_data_offset: %d\n",filter.pkt_data_offset);
printf("filter.pkt_data_size: %d\n",filter.pkt_data_size);
printf("filter.data: ");
for(i=0;i<filter.pkt_data_size;i++)
{
    printf("%02x ",filter.data.b[i]);
}
printf("\nfilter.mask: ");
for(i=0;i<filter.pkt_data_size;i++)
{
```



```
            printf("%02x ",filter.mask.b[i]);
        }
        printf("\n");

        /*删除 id 为 ids[0]的过滤器*/
        if(ids_num != 0)
        if(fsl_rx_filter_destroy(ids[0]))
        {
            printf("destroy filter error\n");
        }
    }
```

11.2 应用层发包

发包时，报文前部有 8 字节的私有包头，携带有包长，目的端口，包优先级，是否经过 PP 模块等信息，供交换芯片内部使用。包头格式如下：

SOP: { start_ind_cfg[31:0], 8'h0, priority[2:0], ppbypass_ind, dport[5:0], pktlen[13:0] }

start_ind_cfg: 可配置，缺省 32'hc704dd7b

```
void trig_tx(pkt_par *pkt_ptr)
{
    u8i tx_buf[256];
    int unit=0;
    fsl_pkt_t pkt;
    mem_clr(tx_buf, 256);  /*包缓冲区清零*/
    pkt.next = NULL;
    pkt.pkt_data = tx_buf;
    pkt.pkt_len = sizeof(tx_buf);
    pkt.port = 0;
    pkt.ppbypass = 1;
```

```
    pkt.priority = 0;

    prepare_pkt(&pkt);    /*设置私有包头信息*/

    fsl_common_tx(unit, &pkt, NULL);    /*发包*/

}
```

11.3 应用层收包

收到的包，包含 8 字节的私有包头和 8 字节的私有包尾。包头携带有包长，源端口，包优先级，是否经过 PP 模块等信息。包尾携带有接收包的时间等信息。

SOP: { start_ind_cfg[31:0], 12'h0, sport[5:0], len[13:0]}

EOP: {end_ind_cfg[31:0] , ts_txout[31:0] }

ts_txout[31:0]: 接收到包的时间。 ts_txout[31:30]为秒的后两位；ts_txout[29:0]为纳秒。

start_ind_cfg: 可配置，缺省 32'hc704dd7b。

end_ind_cfg： 可配置，缺省 32'h004c1db7。

收包处理函数返回值定义如下：

```
typedef enum fsl_rx_e {

    FH_RX_INVALID,        /* 无效包，中止包的处理*/

    FH_RX_NOT_HANDLED,  /* 包未被处理，可以继续被其他包处理函数处理*/

    FH_RX_HANDLED,        /* Handled 计数加 1，还可继续被其他包处理函数处理*/

    FH_RX_HANDLED_OWNED /* owned 计数加 1， 不可继续被其他包处理函数处理*/

} fsl_rx_t;

fsl_rx_t process_pkt(int unit,fsl_pkt_t *pkt,void *pare) /*收包处理函数*/

{

    int i;

    uint8_t *data=pkt->pkt_data;

    printf("pkt(len=%d):\n",pkt->pkt_len);

    for(i=0;i<pkt->pkt_len;i++)

    {
```

```
        printf("0x%02x ",*(data+i));
    }
    printf("\n");
    return FH_RX_NOT_HANDLED; /*该返回值表示包未被处理，还可以被其他处理函数继续处理*/
}
int trig_rx()
{
    if(fsl_common_rx_start(0)!=FSL_ERR_OK)          /*开始接收*/
    {
        printf("start fail\n");
        return -1;
    }
    /*注册收包处理函数，可以注册多个收包处理函数，按照优先级依次处理收到的包。优先级 0 为最低*/
    if(fsl_common_rx_register(0,"print",process_pkt,2,NULL,0)!=FSL_ERR_OK)
    {
        printf("register fail\n");
        fsl_common_rx_shutdown(0);
        return -1;
    }
}
```


12 修订信息

修订时间	版本	描述
2021.5.8	V1.0	初始版本。
2022.12.23	V1.4	内容优化。