life.augmented

AN5683

Application note

SPC58x Ethernet multiple queues and channels

Introduction

The SPC58x microcontroller family embeds advanced 10/100 and Gigabit Ethernet controllers.

This document aims to show how to configure and use the multi-queues and multi-channels embedded in these Ethernet controllers. This document shows the configuration and differences found, about this support, inside all the microcontrollers in the same SPC58x family.

Application is based on SPC58EHx/SPC58NHx microcontroller although the example can be applied for all these microcontrollers considering minor differences.

The Ethernet controllers can support many different configurations to cover several application use-cases. This application note documents a simple example where the untagged traffic is moved among different DMA channels. No performances neither routing mechanism of tagged frames are reported in this document.

1 Ethernet overview

The Ethernet controller is composed by three main functional blocks: the DMA, the MTL which manages the internal FIFOs for each receive and transmit path and the MAC that is responsible for all functional part of the protocol.

The embedded DMA engine has independent transmit and receive hardware. The transmit engine transfers data from system memory to the device port or MAC transaction layer (MTL), while the receive engine transfers data from the device port to the system memory.

The Ethernet controller uses dedicated descriptor lists to efficiently move data from source to destination with minimal CPU intervention. The DMA part can generate interrupts when a packet is received or transmitted. It also notifies all bus error or buffer unavailability events. The MTL block can also notify, for example by interrupts, when overflow conditions are detected on the receive path or when an underflow is detected in the transmit path. The MAC block duty is to configure, manage and generate interrupts linked to PTP protocol settings, MMC counters and remote wakeup low-power modes.

1.1 MTL features

The MTL supports different features for transmit and receive layers, for example:

- Multiple queues on the transmit path with a common memory for all queues
- Store-and-forward mechanism or threshold mode
- Checksum features
- Packet-level control for VLAN and source/destination filtering
- Different scheduling algorithms in configurations for multiple queues

1.2 DMA features

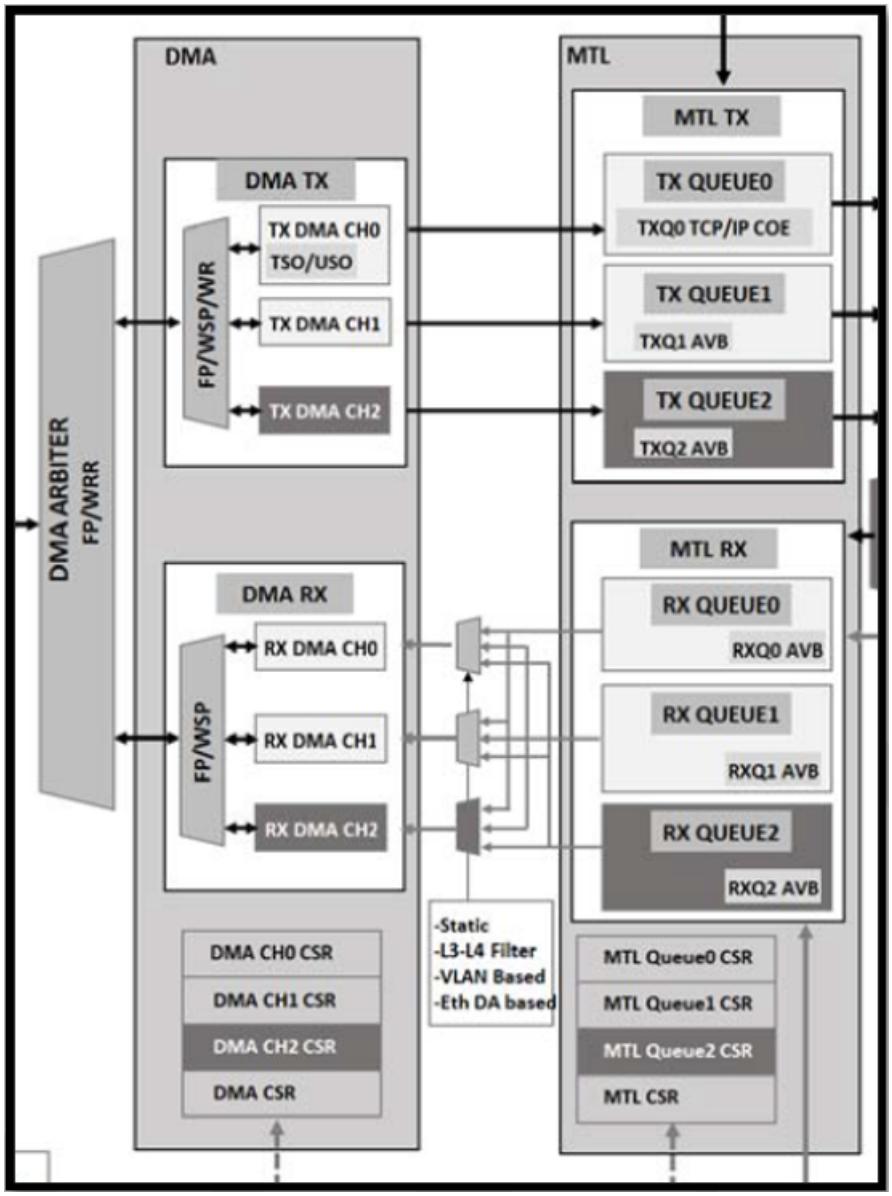
The embedded DMA block supports many features for the transmit and receive paths, for example:

- Multi-channel transmit and receive engines
- Separate DMA channel in the transmit path for each queue in MTL
- Single or multiple DMA channels for any number of queues in MTL receive path
- Programmable interrupt management

The DMA arbiter helps in arbitration of all the paths (transmit and receive) in all channels.

The Figure 1 shows the related DMA and MTL block for the SPC58EHx/SPC58NHx microcontroller. These blocks, related multi-channels and queues management are described in the next Section 2 .

Figure 1. **SPC58EHx/SPC58NHx MTL and DMA blocks**



2 Multiple queues and channels overview

This chapter aims to provide an overview of the multiple queues and channels support, user should refer to the RM0452 reference manual of the device for a complete description.

2.1 DMA multiple channels

2.1.1 Transmit path

When there is any request in the TX queue, the DMA arbiter fetches the descriptor and the priority is assigned following one of these schemes.

Using the default fixed priority (FP) scheme, the channel with highest priority always wins the arbitration when it requests the bus. For example, on SPC58EHx/SPC58NHx MCU, the channel 0 has the lowest priority while the channels 3 has the highest one.

Weighted strict priority (WSP) if any channel does not have a frame to transmit, the weight of that channel gets reassigned to another enabled channel.

Weighted round-robin (WRR), all channels are serviced in round-robin algorithm, the DMA arbiter selects the channel with the highest weight, then the channel with next highest weight, and so on. If any channel does not have a frame to transmit, the weight of that channel gets equally distributed to all channels that have frames to transmit.

The TAA field of the DMA_Mode register is used to select the arbitration algorithm for the transmit side. The TCW field of the DMA_CHn_Tx_Control register provides the weight for each transmit channel as shown in weights for DMA channels.

2.1.2 Receive path

In the receive path, the DMA reads a packet from the MTL receive queue and it writes it to the packet data buffers of the corresponding DMA channel.

Supporting multiple channels, the arbiter considers the data to be transferred according the programmed priority (MTL_RxQ[x]_Control register).

2.1.3 TX and RX DMA priorities

The DMA arbiter also performs the arbitration between the TX and RX paths of DMA channels for accessing descriptors and data buffers. The DMA arbiter supports the fixed priority and weighted round-robin. This is managed by programming the DMA_Mode register fields: DA, PR and TXPR.

In this project the DMA_Mode.DA and TXPR are set to zero and this means that weighted round-robin is used as arbitration schema and TX and RX have equal priority. Anyway, this can be changed by using related configuration structure (refer to next paragraphs).

2.2 MTL queue mapping

2.2.1 Receive queues and channels mapping

On SPC58EHx/SPC58NHx MCU the incoming packets in the MTL RX queues can be routed to any one of the available DMA channels. Two possible configurations are possible:

- Static mapping: an RX queue is connected to a specific DMA channel
- Dynamic mapping: mapping is not static, but it can change for each packet

Moreover, the MAC can assign and route the incoming packets according VLAN tag priority, AV traffic to dedicated queues on some MCU.

In this application the static mapping is used by default.

2.2.2 Transmit arbitration between DMA and MTL

The number of TX DMA channels is always equal to the number of TX queues in the MTL: so, there is one-to-one static mapping for the transmission path.

2.3 MTL multiple queues

The TX queue can be enabled for generic, AV or all types of traffic. Generic transmit queues use WRR or WSP queuing algorithms while the AV TX queue uses CBS or SP queuing algorithms.

The RX queue can be enabled for generic, AV or all types of traffic.

Refer to Section 7.1 for the related programming of the queues adopted in this application.

3 SPC58x queues and channels registers

Each microcontroller in the SPC58x family provides different numbers of internal queues and channels.

The Table 1 and Table 2 show the available queues and channels for each MCU, this can be extracted by the MAC_HW_FEATURE2 read-only register (fields: [R/T]X[CH/Q]CNT + 1).

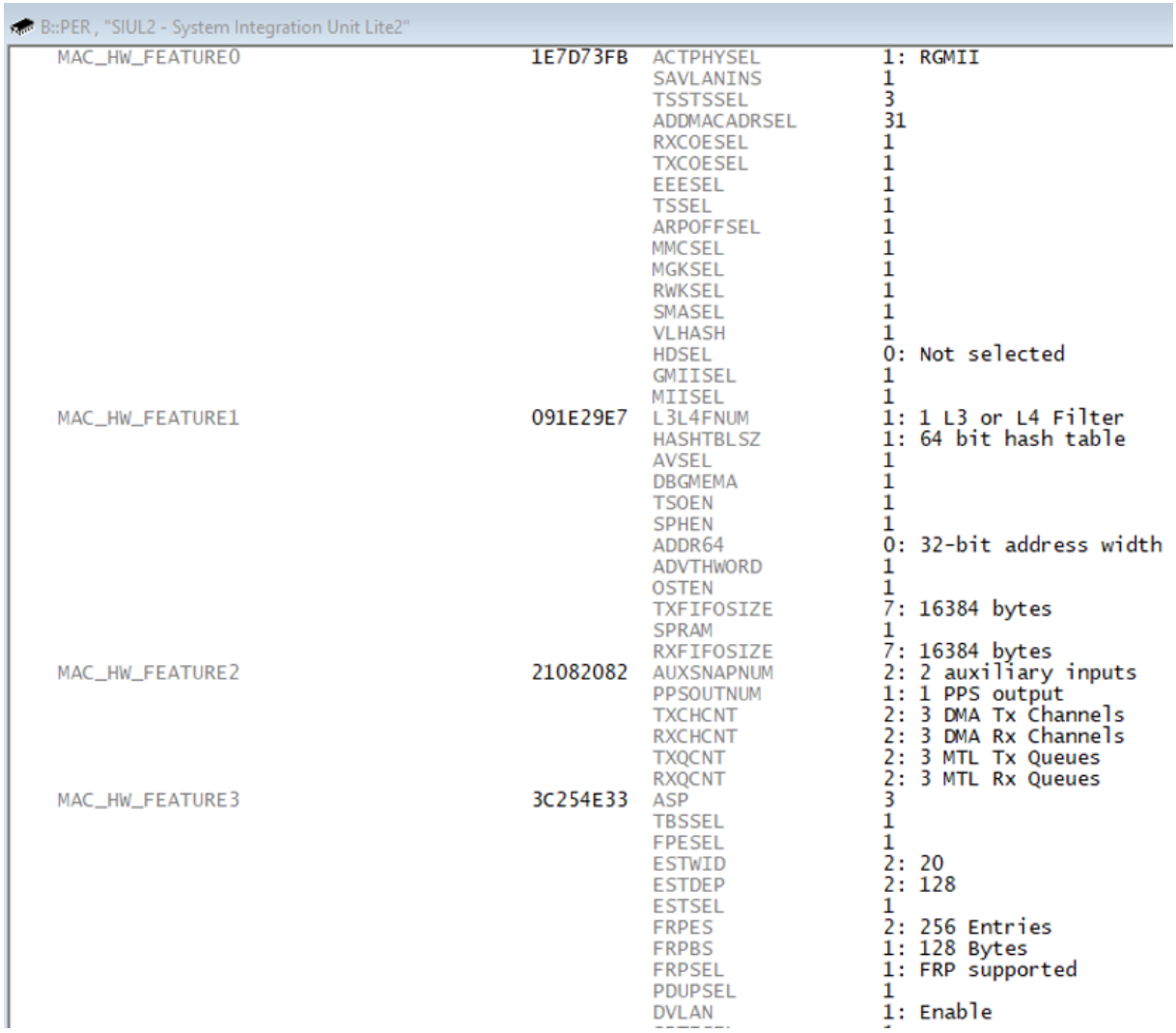
Table 1. SPC58x - number of queues and channels

MCU	Interface	MTL RX queues	DMA RX qhannels	MTL TX queues	DMA TX qhannels
SPC584Bx	ETH0	2	2	2	2
SPC584Cx/SPC58ECx	ETH0	2	2	2	2
SPC58NE84x/SPC58xG84x	ETH0	2	2	2	2
	ETH1	2	2	2	2
SPC58EHx/SPC58NHx	ETH0	2	2	2	2
	ETH1	3	3	3	3

Table 2. SPC58x MAC_HW_FEATURE2 - TX/RX queue and channel fields

MCU	Interface	MTL RX queues	DMA RX channels	MTL TX queues	DMA TX channels
SPC584Bx	ETH0	2	2	2	2
SPC584Cx/SPC58ECx	ETH0	2	2	2	2
SPC58NE84x/SPC58xG84x	ETH0	2	2	2	2
	ETH1	2	2	2	2
SPC58EHx/SPC58NHx	ETH0	2	2	2	2
	ETH1	3	3	3	3

Figure 2. **SPC58EHx/SPC58NHxMAC_HW_FEATURE registers**



3.1 MTL queue registers

3.1.1 Receive and transmit control registers

According to the Ethernet controller embedded in each MCU, there are some register differences that need to be considered before entering in the details of the programming.

Table 3. SPC58x receive queue control register offset

MCU	MAC_RXQ_CTRL0	MAC_RXQ_CTRL1	MAC_RXQ_CTR2	MAC_RXQ_CTRL4
SPC584Bx	0xA0	0xA4	0xA8	N/A
SPC584Cx/SPC58ECx	0xA0	0xA4	0xA8	N/A
SPC58NE84x/SPC58xG84x	0xA0	0xA4	0xA8	N/A
	0xA0	0xA4	0xA8	N/A
SPC58EHx/SPC58NHx	0xA0	0xA4	0xA8	N/A
	0xA0	0xA4	0xA8	0x94

From the Table 3, the MAC_RXQ_CTRL0 register enables the receive queues for AV or generic feature.

The MAC_RXQ_CTRL1 register controls the routing of multicast, broadcast, AV, and untagged packets to the RX queues.

The MAC_RXQ_CTRL2 register controls the routing of tagged packets based on the USP (user priority) field of the received packets to the queues.

The MAC_RXQ_CTRL4 register controls the routing of unicast and multicast packets that fail the destination or source address filter to the RX queues. It is only present on the Gigabit Ethernet controller embedded in the SPC58EHx/SPC58NHx MCU.

3.1.2 Operation mode registers overview

The Table 4 shows the subset of registers that is available for the transmit and receive queues 0.

Table 4. SPC58x MTL queues 0 register set

Register	Description
MTL_Q0_Interrupt_Control_Status	Interrupt enable and status bits for the queue 0 interrupts
Transmit queue 0	
MTL_TxQ0_Operation_Mode	To establish the transmit queue operating modes
MTL_TxQ0_Underflow	This is counter for packets aborted because of transmit underflow
MTL_TxQ0_Debug	It reports the debug status of various blocks related to the transmit
MTL_TxQ0_ETS_Status	The queue 0 ETS status register provides the average traffic
MTL_TxQ0_Quantum_Weight	Quantum value for weighted round-robin
Receive queue 0	
MTL_RxQ0_Operation_Mode	To establish the receive queue operating modes and command
MTL_RxQ0_Missed_Packet_Overflow_Cnt	Missed packet and overflow counter register
MTL_RxQ0_Debug	Receive debug register gives the debug status

While the queues 0 are managed by dedicated register set, the rest of extra queues are managed by the registers summarized in the Table 5.

Table 5. **SPC58x MTL extra queues register set**

Register	Description
MTL_Qn_Interrupt_Control_Status	Interrupt enable and status bits for the queue 0 interrupts
Transmit queue n	
MTL_TxQn_Operation_Mode	To establish the transmit queue operating modes
MTL_TxQn_Underflow	This is counter for packets aborted because of transmit underflow
MTL_TxQn_Debug	It reports the debug status of various blocks related to the transmit
MTL_TxQn_ETS_Control	ETS control register controls the enhanced transmission selection operation.
MTL_TxQn_ETS_Status	ETS status register provides the average traffic
MTL_TxQn_Quantum_Weight	Quantum value for weighted round-robin or idleSlopeCredit in case CBS is activated
MTL_TxQn_Send_slope_Credit	It contains the sendSlope credit
MTL_TxQ(#i)_HiCredit	High credit value required for the credit-based shaper algorithm
MTL_TxQ(#i)_LowCredit	Low credit value required for the credit-based shaper algorithm
Receive queue n	
MTL_RxQn_Operation_Mode	To establish the receive queue operating modes and command
MTL_RxQn_Missed_Packet_Overflow_Cnt	Missed packet and overflow counter register
MTL_RxQn_Debug	Receive debug register gives the debug status

3.2 DMA channel registers overview

The DMA engine is managed by a set of registers summarized in the Table 6 while for other dedicated set of registers to control all the available channels refer to Table 7.

Table 6. **SPC58x DMA registers**

Register	Description
DMA_Mode	To define bus operating modes for the DMA engine: provide reset and common configuration
DMA_SysBus_Mode	To program the bus behavior
DMA_Interrupt_Status	To indicate which event generated the interrupt: MAC, MTL, DMA channel.

Table 7. **SPC58x DMA channel registers**

Register	Description
DMA_Mode	To define bus operating modes for the DMA engine: provide reset and common configuration
DMA_SysBus_Mode	To program the bus behavior
DMA_Interrupt_Status	To indicate which event generated the interrupt: MAC, MTL, DMA channel.

As reported in the RM0452 reference manual of the device and in the example code, provided in the next chapters, other registers are mandatory to program the descriptor list for each DMA channel and getting the related status on transmission and reception processes.

4 Interrupt configuration

All the Ethernet controllers, embedded in this MCU family, have a combined interrupt signal. This means that a common service routine (ISR) can be used for handling all the events from MAC, MTL, MMC and DMA channels. In this case, the ISR must check the status of the related registers to detect which event needs to be serviced. This is possible by looking at dedicated registers.

In case of multiple DMA, for example, the DMA_INTERRUPT_STATUS register shows which channel generate the event and the DMA_CH_STATUS shows exactly the event: e.g. frame reception or transmission completed.

The SPC58EHx/SPC58NHx microcontroller also provides dedicated interrupts for the available transmit and receive channels.

The software application implements the combined interrupt strategy.

The Table 8 summarizes the core and DMA interrupts (also reporting the related numbers).

Table 8. SPC58x MAC core and DMA Interrupts

MCU	Interface	Combined interrupt number	TX CH0	RX CH0	TX CH1	RX CH1	TX CH2	RX CH2
SPC584Bx	ETH0	212	N/A	N/A	N/A	N/A	N/A	N/A
SPC584Cx/SPC58ECx	ETH0	212	N/A	N/A	N/A	N/A	N/A	N/A
SPC58NE84x/SPC58xG84x	ETH0	212	N/A	N/A	N/A	N/A	N/A	N/A
	ETH1	218	N/A	N/A	N/A	N/A	N/A	N/A
SPC58EHx/SPC58NHx	ETH0	212	208	210	209	211	N/A	N/A
	ETH1	218	201	204	202	205	203	206

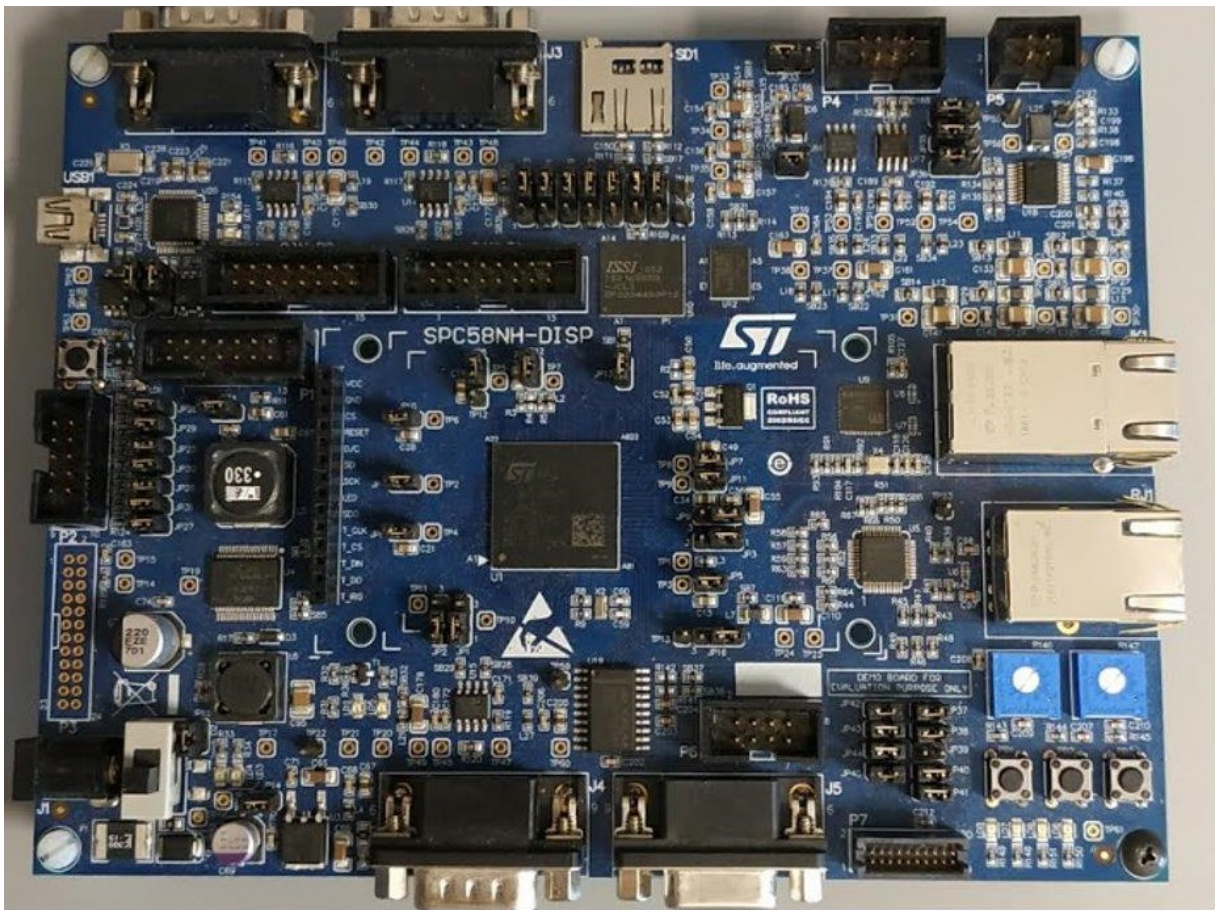
Note: Other interrupts are available for energy efficient Ethernet (LPI) and power management (PMT), not covered by this document.

5 Hardware setup

The example reported in this document has been obtained by using the SPC58EHx/SPC58NHx discovery board and the Ethernet interface 1 which embeds three queues and three channels for both receive and transmit paths.

The Ethernet interface used for testing is wired with a point-to-point connection to an external host machine to generate and verify the network traffic.

Figure 3. SPC58EHx/SPC58NHx discovery board



6 Software overview

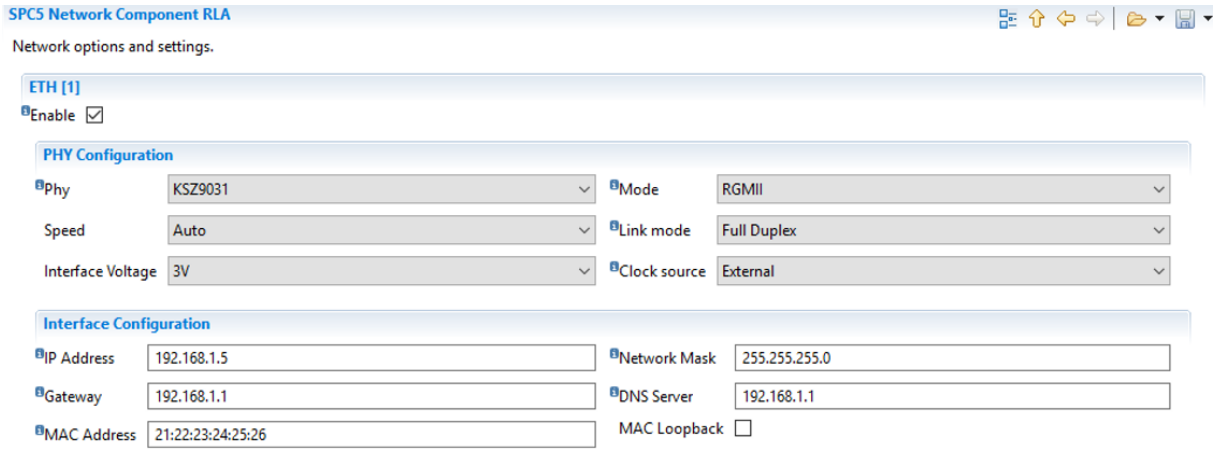
The multiple queues and channels support is built on top of SPC5Studio tool.

By default, SPC5Studio tool provides networking projects for the desired platform where all the related signals of the Ethernet controller, the whole MCU core and the TCP/IP stack are configured.

Expert user can also change the default configuration through graphical interface.

The Figure 4 shows the Ethernet 1 configuration:

Figure 4. SPC5Studio network component view



The example and related programming of the interface also considers the relationship with the RTOS and TCP/IP stack used in this tool.

Note:

1. *Due to the complexity of the whole software infrastructure, this application note just reports the main structures and procedures designed for this specific support.*
2. *For additional information about the networking configuration and internals of the driver refer to the AN5413.*

7 Driver support

This chapter documents the relevant part of the driver to support the multiple channels and queues.

The following function reads the hardware feature registers and stores, in the private structure of the driver, the number of available channels and queues to setup the software according to the real hardware configuration.

```
static void dwmac_qos_hw_features(dwmac_qos_t * priv)
{
    priv->rxcoe = priv->reg->MAC_HW_FEATURE0.B.RXCOESEL;
    priv->mmc = priv->reg->MAC_HW_FEATURE0.B.MMCSEL;
    priv->dma_rx_ch = priv->reg->MAC_HW_FEATURE2.B.RXCHCNT;
    priv->dma_tx_ch = priv->reg->MAC_HW_FEATURE2.B.TXCHCNT;
    priv->dma_tx_queue = priv->reg->MAC_HW_FEATURE2.B.TXQCNT;
    priv->dma_rx_queue = priv->reg->MAC_HW_FEATURE2.B.RXQCNT;
}
```

7.1 MTL queue initialization

The MTL transmit and receive queues have many parameters and several configurations can be applied. This example shows the setup adopted by this driver implementation.

Below the structure passed to the driver to configure all the queues:

```
/* Multiple Queues and Channels */
typedef struct {
    /* Tx Scheduling Algorithm */
    uint32_t tx_algo;
    /* Receive Arbitration Algorithm */
    uint32_t rx_algo;
    /* TX Queue 0 */
    uint32_t tx_q0_ttc;
    uint32_t tx_q0_txqen;
    uint32_t tx_q0_tsf;
    /* RX Queue 0 */
    uint32_t rx_q0_rtc;
    uint32_t rx_q0_rsf;
    /* TX/RX Queues */
    uint8_t  rx_queue_en[MAX_QUEUE];
    uint8_t  rx_queue_prio[MAX_QUEUE];
    uint32_t rx_queue_routing_mask;
    uint8_t  tx_queue_en[MAX_QUEUE];
    uint8_t  tx_queue_weight[MAX_QUEUE];
    uint8_t  rx_queue_weight[MAX_QUEUE];
    /* Priority scheme for Tx DMA and Rx DMA */
    uint32_t rxtx_dma_prio;
    uint32_t rxtx_dma_prio_ratio;
    uint32_t taa;
} mtl_cfg_t;
```

Note: For additional information and register matching with the above fields refer to the communication interface module chapters of RM0452 reference manual of the device.

```
static const mtl_cfg_t mtl_conf = {
    /* Tx Scheduling Algorithm (SCHALG) */
    MTL_MODE_TX_SP,
    /* Receive Arbitration Algorithm */
    /* 0:Strict priority (SP),1: Weighted Strict Priority (WSP) */
    MTL_RX_ALGO_SP,
    /* ----------- TX Queue 0 -----------*/
    0, /* Threshold [32-512] */
    2, /* enable*/
    1, /* Store and forward */
    /* -------- Rx Queues ----------- */
    0, /* Threshold */
    1, /* SF */
    /* Enable -> 1:Q0: AV, 2: -> Q1 and Q2 normal, 0: disabled.*/
    RX_QUEUE_EN_GENERIC, RX_QUEUE_EN_AV, RX_QUEUE_EN_AV, 0, 0, 0, 0, 0,
    RXQ0_PRIO, RXQ1_PRIO, RXQ2_PRIO, 0x0, 0x0, 0x0, 0x0, 0x0,
    0x0, /* Routing mask for control register 1 */
    /* TX Qn enable: 1: AV, 2: normal
     * Note: on some MCU TX_Q0 must be normal by default when enabled */
    TX_QUEUE_EN_NORMAL, TX_QUEUE_EN_AV, TX_QUEUE_EN_AV, 0, 0, 0, 0, 0,
    /* TX queue weights (ISCQW) */
    TXQ0_WEIGHT, TXQ1_WEIGHT, TXQ2_WEIGHT, 0x0, 0x0, 0x0, 0x0, 0x0,
    /* RX queue weights (RXQ_WEGT) */
    RXQ0_WEIGHT, RXQ1_WEIGHT, RXQ2_WEIGHT, 0x0, 0x0, 0x0, 0x0, 0x0,
    /* Priority scheme for Tx DMA and Rx DMA */
    DMA_TX_RX_EQUAL_PRIO,
    1,
    DMA_MODE_WRR
};
```

During the driver initialization phase the following function is invoked to configure both queue0 (Q0) and the extra queues (Qn) for TX and RX paths.

```
static void dwmac_qos_mtl_init(dwmac_qos_t * priv) {
    uint32_t tx_fifosize, rx_fifosize;
    /* MTL mode and Queue 0 configuration */
     /* Tx Scheduling Algorithm */
    priv->reg->MTL_OPERATION_MODE.B.SCHALG = priv->mtl_param->tx_algo;
    /* Weighted Round Robin */
    if (priv->mtl_param->tx_algo == MTL_MODE_TX_WRR) {
        dwmac_qos_tx_queue_weight(priv);
    }
    /* Receive Arbitration Algorithm */
    priv->reg->MTL_OPERATION_MODE.B.RAA = priv->mtl_param->rx_algo;
    dwmac_qos_rx_queue_weight(priv);
    priv->reg->MTL_TXQ0_OPERATION_MODE.B.TSF = priv->mtl_param->tx_q0_tsf;
    priv->reg->MTL_TXQ0_OPERATION_MODE.B.TTC = priv->mtl_param->tx_q0_ttc;
    priv->reg->MTL_TXQ0_OPERATION_MODE.B.TXQEN = priv->mtl_param->tx_q0_txqen;
    priv->reg->MTL_RXQ0_OPERATION_MODE.B.RSF = priv->mtl_param->rx_q0_rsf;
    priv->reg->MTL_RXQ0_OPERATION_MODE.B.RTC = priv->mtl_param->rx_q0_rtc;
    /* Queue sizes are calculated by using HW feature registers */
    tx_fifosize = 128 << priv->reg->MAC_HW_FEATURE1.B.TXFIFOSIZE;
    tx_fifosize = tx_fifosize / priv->dma_channel;
    tx_fifosize = tx_fifosize / 256U - 1;
    priv->reg->MTL_TXQ0_OPERATION_MODE.B.TQS = tx_fifosize;
    rx_fifosize = 128 << priv->reg->MAC_HW_FEATURE1.B.RXFIFOSIZE;
    rx_fifosize = rx_fifosize / priv->dma_channel;
    rx_fifosize = rx_fifosize / 256U - 1;
    priv->reg->MTL_RXQ0_OPERATION_MODE.B.RQS = rx_fifosize;
    /* Enable and configure the RX/TX queues */
    dwmac_qos_rx_queue(priv, true, rx_fifosize);
    dwmac_qos_tx_queue(priv, tx_fifosize);
}
```

As shown above, the software must ensure that the sum of each TX and RX queue size should not exceed the values (TXFIFOSIZE and RXFIFOSIZE) inside the MAC_HW_FEATURE1 registers.

The dwmac_qos_tx_queue_weight function is showed as example to program the ISCQW fields in the MTL_TXQ0_QUANTUM_WEIGHT and MTL_TXQn_QUANTUM_WEIGHT registers with the values fixed in the mtl_param->tx_queue_weight by the user. Similar function is implemented to program the RXQ_WEGT fields in the MTL_RXQ0_CONTROL and MTL_RXQn_CONTROL registers.

Then the dwmac_qos_rx_queue and dwmac_qos_tx_queue program the extra queues as shown in the next paragraph.

Note: The AN5413 documents the internal private structure of the driver where the mtl_param is now added to manage FIFO and queues.

Below a snapshot of the structure to see the main fields:

```
/* Driver private structure */
typedef struct {
    /* Pointer to the TX and RX queues (1:1 match with channels) */
    dwmac_qos_q_c_t ch[MAX_QUEUE];
    uint32_t tx_len;         /* TX ring size */
    uint32_t rx_len;         /* RX ring size */
    uint32_t bufsize;        /* Buffer size */
    uint32_t mode;           /* Interface mode */
    unsigned int rx_handler_bad_frame; /* Error in RX desc */
    bool rxcoe;            /* RX csum */
    bool txcoe;            /* TX csum */
    /* Number of available channels and queues from * MAC_HW_FEATURE2 */
    uint32_t dma_tx_ch;
    uint32_t dma_tx_queue;
    uint32_t dma_rx_ch;
    uint32_t dma_rx_queue;
    uint32_t dma_channel;
    bool rx_irq_en;        /* Enable IRQ on RI (RWT not used) */
    bool rwt_en;           /* RX WDT enable */
    bool mmc;              /* Has MAC counters */
    uint32_t pmt;          /* Has Power Management block */
    bool lpi;              /* Has LPI */
    bool timestamp;        /* Has TimeStamp block */
    bool loopback;         /* Enable the Internal MAC loopback */
    dwmac_stats_t desc_status;       /* Save Desc status info */
    dma_stats_t dma_irq_stat;        /* Save DMA IRQ status */
    core_irq_stat_t core_irq_stat;   /* Save MAC Core IRQ status */
    mtl_cfg_t *mtl_param;            /* MTL parameters */
…
volatile spc5_eth *reg;        /* ETHERNET_1_tag or ETHERNET_0_tag */
…
} dwmac_qos_t;
```

7.1.1 Enabling MTL queues

The following code is used to manage the RX queues in the MTL layer. These can be configured for either AV/B or normal mode. The driver runs on all MCUs in the SPC58x family and the number of queues, as detailed in the previous chapters, can change. So, the following functions can run on all the configuration according to the number of queues available.

The configuration must ensure that the queue priorities (PSRQn) are mutually exclusive, all weights must be assigned to both TX and RX paths; this is to guarantee the correct routing according on selected algorithm. For example, the TXQ0,1,2_WEIGHT, RXQ0,1,2_WEIGHT and RXQ0,1,2_PRIO defined above can be configured according the application use-case.

```
static void dwmac_qos_rx_queue(dwmac_qos_t *priv, bool enable, uint32_t fifosize) {
    int i;
    uint32_t value = 0, prio = 0;
    uint32_t reg_value, offset, match = 0;
    void *p = priv->reg;
    if (enable) {
         /* Extra queue mode configuration: only set the fifo size
          * and the store-and-forward as default modes; like the Q0.
          * --> Qn are configured as well as the Q0 */
        for (i = 0; i <= priv->dma_rx_queue; i++) {
             /* Only program the MTL_RXQn_OPERATION_MODE register
              * (use the offset to cover different platforms)
              * Do not program other fields for flow control
              * and other frame passing features. */
            if (i > 0) {
                offset = (RX_QN_OFFSET + (i * RX_QN_OPERATION_MODE_OFFSET));
                reg_value = ((fifosize << RX_FIFO_RQS_SHIFT)
                 |(priv->mtl_param->rx_q0_rsf << RX_FIFO_RSF_SHIFT)
                 |(priv->mtl_param->rx_q0_rtc << RX_FIFO_RTC_SHIFT));
                  /* Number of queues can change according to the MCU
                   * calculate at runtime the offset and program the
                   * register w/o using union structure. */
                REG_WRITE32((p + offset), reg_value);
            }
             /* Enable the Qn for either AV/B or normal mode. */
            value |= (priv->mtl_param->rx_queue_en[i] << (i * 2));
            prio |= priv->mtl_param->rx_queue_prio[i] << (i * 8);
            /* MTL Queue and DMA channel assignment: Channel<x> - Queue<x> */
            match |= (i << (i * 8));
        }
        priv->reg->MAC_RXQ_CTRL0.R = value;
        priv->reg->MAC_RXQ_CTRL2.R = prio;
        priv->reg->MAC_RXQ_CTRL1.R = priv->mtl_param->rx_queue_routing_mask;
        priv->reg->MTL_RXQ_DMA_MAP0.R = match;
    } else {
        priv->reg->MAC_RXQ_CTRL0.R = 0;
        priv->reg->MAC_RXQ_CTRL2.R = 0;
        priv->reg->MAC_RXQ_CTRL1.R = 0;
    }
}

static void dwmac_qos_tx_queue(dwmac_qos_t *priv, uint32_t fifosize) {
    int i;
    uint32_t reg_value, offset;
    void *p = priv->reg;

    for (i = 1; i <= priv->dma_tx_queue; i++) {
        offset = (TX_QN_OFFSET + (i * TX_QN_OPERATION_MODE_OFFSET));
            /* Configure the Qn the SF and threshold.
             * Also enable as AV or normal mode
             * the extra Queues because the Q0 is normal. */
        reg_value = ((fifosize << TX_FIFO_RQS_SHIFT)
            | (priv->mtl_param->tx_q0_tsf << TX_FIFO_TSF_SHIFT)
            | (priv->mtl_param->tx_q0_ttc << TX_FIFO_TTC_SHIFT)
            | (priv->mtl_param->tx_queue_en[i] <<TX_FIFO_TQE_SHIFT));
        REG_WRITE32((p + offset), reg_value);
    }
}
```

The Figure 5 and Figure 6 show a snapshot of the registers programmed by the example functions above.

Figure 5. MTL queue 0 programming

MAC_RXQ_CTRL0	00000029	RXQ2EN	2: Queue 1 enabled for generic
		RXQ1EN	2: Queue 1 enabled for generic
		RXQ0EN	1: Queue 0 enabled for AV
MAC_RXQ_CTRL1	00002000	FPRQ	0
		TPQC	0
		TACPQE	0
		MCBCQEN	0
		MCBCQ	0: Rx Queue 0
		UPQ	2: Rx Queue 2
		PTPQ	0: Rx Queue 0
		AVCPQ	0: Rx Queue 0
MAC_RXQ_CTRL2	00000000	PSRQ2	00
		PSRQ1	00
		PSRQ0	00
MTL_TXQ0_OPERATION_MODE	003F000A	TQS	3F
		TTC	0: 32
		TXQEN	2: Enabled
		TSF	1
MTL_RXQ0_OPERATION_MODE	03F00020	RQS	3F
		RFD	0
		RFA	0
		EHFC	0
		DIS_TCP_EF	0
		RSF	1
		FEP	0
		FUP	0
		RTC	0: 64

Figure 6. MTL queue 1 programming

MTL_TXQ1_OPERATION_MODE	003F000A	TQS	3F
		TTC	0: 32
		TXQEN	2: Enabled
		TSF	1
		FTQ	0
MTL_TXQ1_UNDERFLOW	00000000	UFCNTOVF	0
		UFFRMCNT	0000
MTL_TXQ1_DEBUG	00000000	STXSTSF	0
		PTXQ	0
		TXSTSFSTS	0
		TXQSTS	0
		TWCSTS	0
		TRCSTS	0: Idle state
		TXQPAUSED	0
MTL_TXQ1_ETS_CONTROL	00000000	SLC	0: 1 Slot
		CC	0
		AVALG	0
MTL_TXQ1_ETS_STATUS	00000000	ABS	000000
MTL_TXQ1_QUANTUM_WEIGHT	00000000	ISCQW	000000
MTL_TXQ1_SENDSLOPECREDIT	00000000	SSC	0000
MTL_TXQ1_HICREDIT	00000000	HC	00000000
MTL_TXQ1_LOCREDIT	00000000	LC	00000000
MTL_Q1_INTERRUPT_CONTROL_STATUS	00000000	RXOIE	0
		RXOVFIS	0
		ABPSIE	0
		TXUIE	0
		ABPSIS	0
		TXUNFIS	0
MTL_RXQ1_OPERATION_MODE	03F00020	RQS	3F
		RFD	0
		RFA	0
		EHFC	0
		DIS_TCP_EF	0
		RSF	1
		FEP	0
		FUP	0
		RTC	0: 64

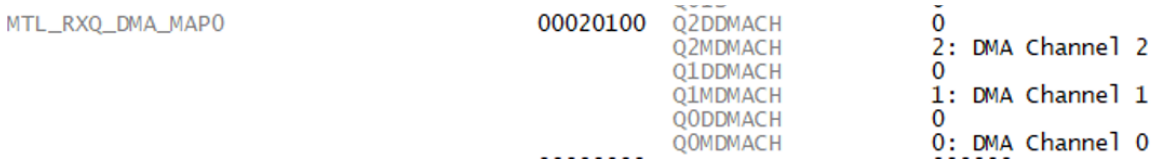
7.2 DMA channel initialization

The example code below is invoked during the initialization to configure each available DMA channel. It enables the interrupts and sets the head of each descriptor ring.

```
static void dwmac_qos_dma_ch_init(dwmac_qos_t * priv, unsigned int channel)
{
    uint32_t desc_addr;

    /* Mask IRQs */
    priv->reg->DMA_CH[channel].DMA_CH_INTERRUPT_ENABLE.B.NIE = 1;
    priv->reg->DMA_CH[channel].DMA_CH_INTERRUPT_ENABLE.B.AIE = 1;
    priv->reg->DMA_CH[channel].DMA_CH_INTERRUPT_ENABLE.B.FBEE = 1;
    priv->reg->DMA_CH[channel].DMA_CH_INTERRUPT_ENABLE.B.RIE = 1;
    priv->reg->DMA_CH[channel].DMA_CH_INTERRUPT_ENABLE.B.TIE = 1;
    priv->reg->DMA_CH[channel].DMA_CH_INTERRUPT_ENABLE.B.RBUE = 1;

    /* Set PBL: FIXME: check validation values */
    priv->reg->DMA_CH[channel].DMA_CH_CONTROL.B.PBLX8 = 1;
    priv->reg->DMA_CH[channel].DMA_CH_TX_CONTROL.B.TXPBL = 8;
    priv->reg->DMA_CH[channel].DMA_CH_RX_CONTROL.B.RXPBL = 8;

    priv->reg->DMA_CH[channel].DMA_CH_TX_CONTROL.B.OSF = 1;
    priv->reg->DMA_CH[channel].DMA_CH_RX_CONTROL.B.RBSZ = SET_RBSZ(BUF_SIZE);
    /* Program the descriptors */
    desc_addr = (priv->ch[channel].dma_tx + (channel * priv->tx_len));
    priv->reg->DMA_CH[channel].DMA_CH_TXDESC_LIST_ADDRESS.R = (uint32_t) desc_addr;
    desc_addr = (priv->ch[channel].dma_rx + (channel * priv->rx_len));
    priv->reg->DMA_CH[channel].DMA_CH_RXDESC_LIST_ADDRESS.R = (uint32_t) desc_addr;
}
```

The Figure 7 shows the matching between DMA channels and MTL queues adopted in this project.

Figure 7. MTL queue and DMA channel matching



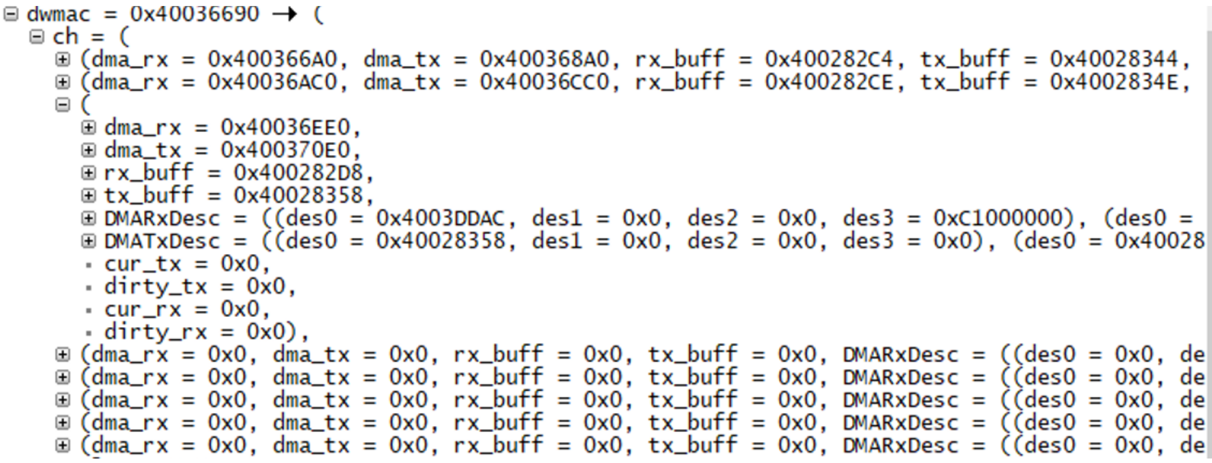
Below the routine used to assign the TX and RX DMA channel priority. By default, the RX and TX channels have the same priority.

```
static void dwmac_tx_rx_dma_prio(dwmac_qos_t * priv)
{
    priv->reg->DMA_MODE.B.TAA = priv->mtl_param->taa;

    if (priv->mtl_param->rxtx_dma_prio == DMA_TX_RX_EQUAL_PRIO) {
        priv->reg->DMA_MODE.B.DA = 0;
        priv->reg->DMA_MODE.B.TXPR = 0;
    } else if  (priv->mtl_param->rxtx_dma_prio == DMA_TX_PRIO_OVER_RX) {
        priv->reg->DMA_MODE.B.DA = 0;
        priv->reg->DMA_MODE.B.TXPR = 1;
    } else {
        priv->reg->DMA_MODE.B.DA = 1;
        priv->reg->DMA_MODE.B.TXPR = 0;
    }

    priv->reg->DMA_MODE.B.PR = DMA_MODE_PR_x_1(priv->mtl_param->rxtx_dma_prio_ratio);
}
```

7.3 Resource allocation

When initializing the driver resources, the application gets from the hardware feature register the number of available queues and channels to configure on the fly the device and to allocate the resources for the drivers. This means, the initialization applies a default configuration for all the queues and channels in terms of algorithm used for scheduling, priorities and mapping between channels and queues. The software must allocate the descriptor rings for all the DMA channels and initializes the related pointers and counters before starting the transmit and receive processes.

Figure 8. Internal driver channels



The Figure 8 shows a typical layout of the transmit and receive rings allocated when three DMA channels are available for each path.

The Figure 9 aims to show a snapshot of the rings in the memory.

All the descriptors inside the rings are programmed in the same way.

Figure 9. **Snapshot of TX and RX rings and buffers**

Note: To keep the compatibility with the FreeRTOS and related TCP/IP stack configuration inside SPC5Studio, two single rings are allocated since the beginning by the tool, these are for transmission and reception paths. These rings are virtually partitioned to implement separated rings for all the DMA channels available. This means that each channel uses a portion of the pre-allocated ring simplifying the implementation and covering all the needs of this application designed to demonstrate the feature.

The example code below shows a portion of the driver used to initialize the rings.

It is called in a loop for all the available channels. This function allocates the buffers and invokes the low-level driver to setup the descriptors.

```
static void dwmac_init_rings(dwmac_qos_t * dwmac, unsigned int channel)
{
    dma_desc_t *p;
    int i;
    uint32_t *rx_buff;
#if ( ipconfigZERO_COPY_TX_DRIVER == 0 )
    uint8_t *tx_buff;

    tx_buff = dwmac->ch[channel].tx_buff;
#endif
    /* Init the descriptors */
    dwmac->ch[channel].dma_rx = dwmac->ch[channel].DMARxDesc;
    dwmac->ch[channel].dma_tx = dwmac->ch[channel].DMATxDesc;

    /* Init the RX ring for this DMA channel */
    p = dwmac->ch[channel].dma_rx + (channel * dwmac->rx_len);
    rx_buff = (uint32_t *) dwmac->ch[channel].rx_buff;
    for (i = 0; i < dwmac->rx_len; i++) {
        dwmac_alloc_rx_buffer(rx_buff);
        p->des0 = (uint32_t) * rx_buff;
        dwmac->descs->init_rx_desc(p, dwmac->rx_irq_en);

        p++;
        rx_buff++;
    }

    /* Init the TX ring for this DMA channel */
    p = dwmac->ch[channel].dma_tx + (channel * dwmac->tx_len);
    for (i = 0; i < dwmac->tx_len; i++) {
        dwmac->descs->init_tx_desc(p);

#if ( ipconfigZERO_COPY_TX_DRIVER == 0 )
        p->des0 = (uint32_t) tx_buff;
        tx_buff += BUF_SIZE;
#endif
        p++;
    }
}
```

Note:

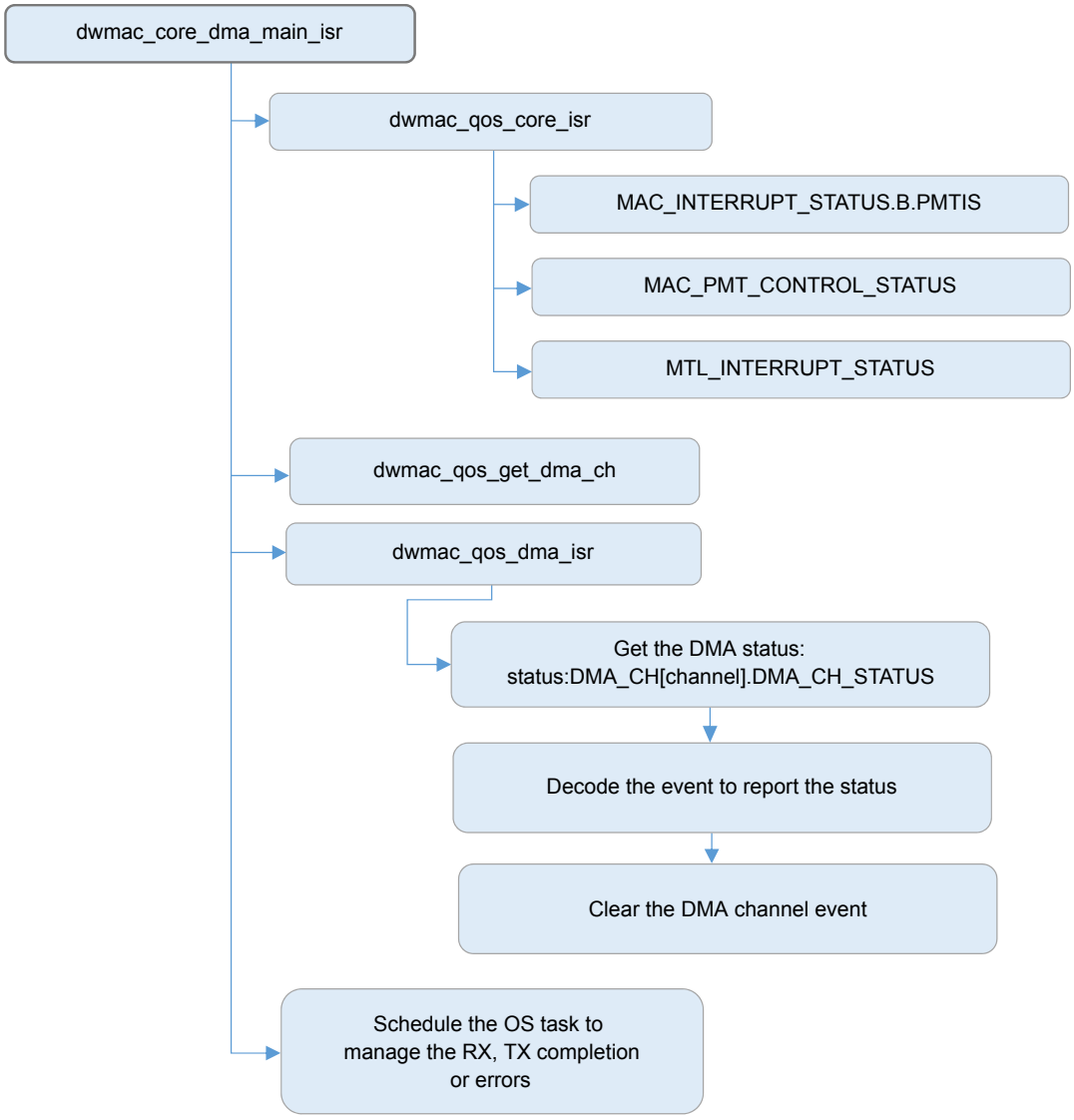
1. *The zero-copy mechanism is not used in this example. The ipconfigZERO_COPY_TX_DRIVER is found inside the FreeRTOS configuration files.*
2. *The zero-copy approach is used for moving the buffers w/o performing any memory copy. This kind of support needs to be implemented and enabled inside the FreeRTOS TCP/IP stack.*
3. *For low level driver API, refer to the AN5413.*

7.4 Interrupt handling

This is an example of the Interrupt service routine defined by SPC5Studio application for the Ethernet interface. Refer to SPC5Studio related documentation for more details about IRQ internals and related APIs.

```
IRQ_HANDLER(SPC5_ETH1_CORE_HANDLER)
{
    IRQ_PROLOGUE();
    dwmac_core_dma_main_isr(&drv_instance[SPC5_ETH1_INSTANCE]);
    IRQ_EPILOGUE();
}
```

Figure 10. ISR block schema (combined IRQ)



The schema shown in the Figure 10 details the flow of the main ISR: dwmac_core_dma_main_isr.

It combines all the DMA and MAC interrupts as detailed in the previous Section 4 Interrupt configuration.

In a glance, the dwmac_core_dma_main_isr checks the MAC interrupts and then invokes the dwmac_qos_get_dma_ch routine. It is used to understand which DMA channel has generated the event. This interrupt status is cleared as soon as the DMA channel status is rewritten.

```
static unsigned int dwmac_qos_get_dma_ch(dwmac_qos_t * priv)
{
    unsigned int channels;
    unsigned int dma_status;

    channels = priv->reg->DMA_INTERRUPT_STATUS.R;
    channels &= DMA_EVENT_STATUS_MASK;

    return channels;
}
```

Note: The dwmac_qos_get_dma_ch could not be used in case of the DMA channels are managed by own separated interrupt ISR (feasible in SPC58EHx/SPC58NHx MCU).

The dwmac_core_dma_main_isr invokes the dwmac_qos_dma_isr to check what is the DMA event, for example: a new frame is in the channel ring or a transmission has been completed.

This software design is based on operating system task that is scheduled by the ISR. This task services the receive process getting all the incoming frames for the DMA rings and passing them to the upper-layer TCP/IP stack. This task sanitizes the ring in case of unexpected errors and it reinitializes the transmission descriptor if the frame has been successfully transmitted.

7.5 Testing the DMA channels

To test the transmission, in this example, the xNetworkInterfaceOutput has been modified to invoke the low-level driver passing the desired channel.

```
netd->netdrv.prepare_send(netd->netdrv.priv,
                  pxDescriptor->pucEthernetBuffer,
                  ulTransmitSize, channel);
```

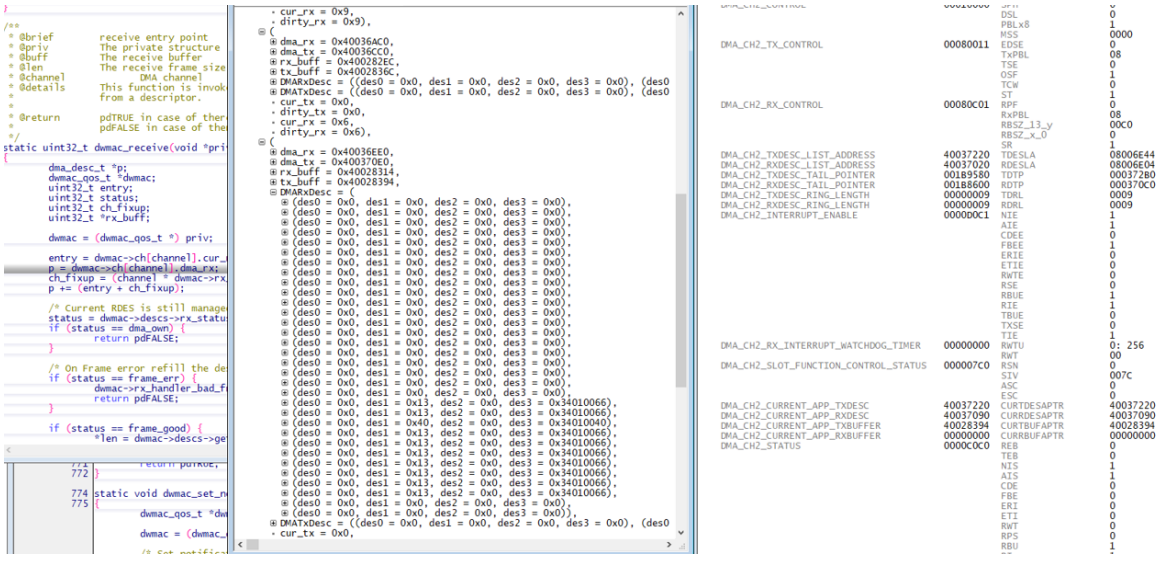
Outcoming frames is moved through the channel passed to the driver prepare_send function. As soon as the frame is sent, the related interrupt occurs notifying the status.

To stimulate the receive process to be moved on different DMA channels, the driver has been instrumented to route untagged frame on desired channel.

In fact, the untagged RX packets can be routed based on the RX queue number specified in the UPQ field of MAC_RXQ_CTRL1 register and the corresponding RX Queue is enabled through RXQ#EN field in MAC_RXQ_CTRL0 register.

The Figure 11 shows a debug session where the incoming frames are moved by the DMA channel 2.

Figure 11. Receiving frames on DMA channel 2



Note: By default, the channels 0 are always used.

Appendix A Acronyms, abbreviations and reference documents

Table 9. Acronyms and abbreviations

Terms	Description
API	Application programming interface
FP	Fixed priority
IP	Internet protocol
ISR	Interrupt service routine
MAC	Media access control
MTL	Transaction layer
TCP	Transmission control protocol
UDP	User datagram protocol
VLAN	Virtual local area network
WRR	Weighted round-robin
WSP	Weighted strict priority

Table 10. Reference documents

Document name	Document title
RM0452	SPC58 H Line - 32 bit Power Architecture automotive MCU Triple z4 cores 200 MHz, 10 MBytes Flash, HSM, ASIL-D
TN1257	SPC58EHx, SPC58NHx IO definition: signal description and input multiplexing tables
AN5413	Getting started with the SPC58x Networking

Revision history

Table 11. **Document revision history**

Date	Revision	Changes
20-Jul-2021	1	Initial release.

Contents

List of tables

List of figures