****

20450 Century Boulevard

Germantown, MD 20874

**EMAC Low Level Driver**

**Software Design Document**

Revision 1.3

September 6, 2019

**Document License**

This work is licensed under the Creative Commons Attribution-Share Alike 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

**Contributors to this document**

Copyright (C) 2010-2019 Texas Instruments Incorporated - <http://www.ti.com/>

|  |  |  |
| --- | --- | --- |
|  | Revision Record | |
|  | Document Title: **Software Design Specification** | |
| Date | Revision | Description of Change |
| 04-04-2017 | 0.1 | Initial Release |
| 04-10-2017 | 0.2 | Added feature list |
| 07-07-2017 | 0.3 | Addressed all the peer review comments |
| 06-13-2018 | 0.3 | Added feature list for AM65XX |
| 07-13-2018 | 0.4 | Added driver details for AM65xx |
| 10-04-2018 | 0.5 | Added document license for Creative Commons |
| 10-05-2018 | 0.6 | WIP draft Incorporated subset of review comments |
| 11-27-2018 | 0.7 | Added sections for IOCTL commands, driver initialization , TX Time Stamp, TX Software Descriptor Return Queue Processing, RX Time Stamp |
| 02-21-2019 | 0.8 | Incorporate follow up comments from customer |
| 3-28-2019 | 0.9 | Incorporate additional follow up comments from customer |
| 4-2-2019 | .95 | Additional IOCTL updates |
| 5-08-2019 | 1.0 | Updates to error types, Clarification of each IOCTL type to be either synchronous versus asynchronous, adding details for EMAC\_IOCTL\_INTERFACE\_MAC\_CONFIG |
| 5-15-2019 | 1.1 | IOCTL table updates.  Enhanced emac poll link status section.  Enhanced Packet RX section.  Miscellaneous updates. |
| 5-31-2019 | 1.2 | Added sections 7.4.2.1, 7.4.2.1.1 and 7.4.2.2  Added clarification to emac\_poll\_ctrl API and ring bitmap.  Updated FDB ADD IOCTL details |
| 9-6-2019 | 1.3 | Section 7.4.3: Updated clarification of return codes to application during emac\_send  Section 7.4.5: Further clarification to TX timestamp implementation  Section 7.4.6: Added details of how ring number is returned in rx callback |

**TABLE OF CONTENTS**

1. Purpose 1

2. Functional Overview 1

3. Assumptions 1

4. Definitions, Abbreviations, Acronyms 1

5. References 1

6. Design Constraints 2

6.1 External Constraints / Features 2

6.2 External Constraints / System Performance 2

6.3 Internal Constraints / Requirements 2

7. System Overview 2

7.1 System Context 2

7.2 Functional Description 3

7.3 CPPI Based IP Driver: IP Version 0/1/4 4

**7.3.1** EMAC Peripheral configuration 4

**7.3.2** Queue Management 5

**7.3.3** Packet Descriptor 5

**7.3.4** Packet TX 5

**7.3.5** Packet RX 6

**7.3.6** Single Critical Section 6

**7.3.7** Multi-core Critical section 6

**7.3.8** Interrupts 7

7.4 UDMA/NavSS based IP Driver: IP Version 5 7

**7.4.1** Memory 8

**7.4.2** EMAC Driver Initialization configuration 9

7.4.2.1 EMAC Open Configuration Details. 10

7.4.2.2 ICSSG Port Queue Configuration 15

**7.4.3** Packet TX 15

**7.4.4** TX Software Descriptor Return Queue Processing 16

**7.4.5** TX Time Stamp 17

**7.4.6** Packet RX 18

**7.4.7** RX Time Stamp 18

**7.4.8** New APIs 19

**7.4.9** IOCTL API Details 21

14.4.9.1 Switch use case 23

14.4.9.2 DUAL MAC Use Case (Single Instance ICSS FW) 30

14.4.9.3 DUAL MAC Use Case (Interposer Instance ICSS FW) 32

Placeholder for future development. 32

**7.4.10** Platform Specific functions/configuration 32

**7.4.11** Interrupts 32

**7.4.12** Multi- Core Support 32

**7.4.13** Interposer Card Support 32

14.4.9.4 Switch Use Use Case (Switch F/W for Interposer Card) 33

14.4.9.5 Dual MAC Use Case (Interposer Card with Standalone Dual Mac F/W) 33

7.5 EMAC Polling Link Status 34

7.6 Error Handling 35

8. Standards, Conventions and Procedures 36

8.1 Documentation Standards 36

8.2 Naming conventions 36

8.3 Programming Standards 36

8.4 Software development tools 36

9. IP Feature List Comparison 37

10. System Design 42

10.1 Design Approach 42

10.2 Dependencies 42

10.3 Decomposition of System 43

**10.3.1** Platform Independent APIs 43

**10.3.2** Platform specific functions/configurations 43

**10.3.3** Operating System Abstraction Layer (OSAL) 43

**10.3.4** CSL Functional Layer 43

**10.3.5** CSL Register Layer 44

11. OMAPL13x Integration 44

11.1 Platform Independent API 44

11.2 Platform Specific functions/configuration 44

11.3 OSAL 44

11.4 CSL 44

11.5 Build Setup 44

# Purpose

This document describes the functionality, architecture, and operation of the Ethernet Media Access Controller (EMAC) Low Level Driver. Also the data types, data structures and application programming interfaces (API) provided by the EMAC driver are explained in this document.

# Functional Overview

EMAC driver provides a well-defined API layer which allows applications to use the EMAC peripheral to control the flow of packet data from the processor to the PHY and the MDIO module to control PHY configuration and status monitoring.

# Assumptions

NA

# Definitions, Abbreviations, Acronyms

| **Term** | **Description** |
| --- | --- |
| API | Application Programming Interface |
| CSL | Chip Support Library |
| EMAC | Ethernet Media Access Controller |
| LLD | Low Level Driver Design |
| ISR | Interrupt Service Routine |
| MDIO | Managed Data Input Output |
| MMR | Memory Mapped Registers |
| NDK | Network Development Kit |
| NIMU | Network Interface Management Unit |
| OSAL | Operating System Adaptation Layer |
| PHY | Physical layer |
| MGMT | Management |
| FW | Firmware |

Table 1 : Abbreviations and acronyms

# References

Following references are related to the features described in this document and shall be consulted as necessary.

* TRM for SoCs being supported by EMAC LLD
* Migrating\_Applications\_from\_EDMA\_to\_UDMA\_using\_TI-RTOS.pdf (ti/drv/udma/docs)

# Design Constraints

## External Constraints / Features

* EMAC LLD should access OS components only through OSAL.

## External Constraints / System Performance

EMAC LLD should allow applications to transfer and receive through Ethernet port and communicate with the network devices at maximum possible speed as supported by HW.

## Internal Constraints / Requirements

EMAC LLD should use CSL layer for register access to abstract the HW dependencies and maintain portability across the platforms.

# System Overview

## System Context

EMAC LLD is designed to be functional as part of TI processor SDK driver package. There will be several components in the processor SDK, apart from applications, which uses EMAC LLD. Driver design ensures that it fits into system properly and provides suitable APIs for utilizing EMAC HW functionality.

The following figure shows the architecture of processor SDK sub-system around the LLD modules.



Figure 1 : Process SDK driver subsystem architecture

## Functional Description

The EMAC driver is responsible for the following:-

* EMAC/MDIO configuration & Queue Management
* Providing a well-defined API to interface with the applications
* Well defined operating system adaptation layer API which supports single core and multiple core critical section protection

The next couple of sections document each of the above mentioned responsibilities in greater detail:

## CPPI Based IP Driver: IP Version 0/1/4

### EMAC Peripheral configuration

The EMAC driver test application provides a sample implementation sequence which initializes and configures the EMAC IP block. This implementation is sample only and application developers are recommended to modify it as deemed fit.

The initialization sequence is not a part of the EMAC driver library. This was done because the EMAC initialization sequence has to be modified and customized by application developers.

The following figure shows the EMAC API the application can call to initialize the EMAC peripheral:-

Figure 2 : EMAC configuration



Please note that the call flow dedicated above is basic illustration of how *emac\_open* is handled internally and may differ from amongst different IP versions of the driver. At the API level from application point of view, it’s the same.

Refer to the EMAC\_OPEN\_CONFIG\_INFO\_T as defined in emac\_drv.h for details of configuration parameters passed into the driver at the time of *emac\_open* API call.

When this API is called, the EMAC driver will first initialize common EMAC configurations (e.g. loopback mode, MDIO enable, PHY address, packet size, etc.) which applies to all the cores, and then initialize the core specific configurations (e.g. channel/MAC address configuration, TX/RX packet descriptor queue size, call back functions, etc.). The driver may also need to do some device specific configurations (e.g. C6457 & C6474 have a SGMII interface in the EMAC peripheral which need to be configured, and C6474 has a hardware semaphore which also needs to be configured).

The *emac\_config*() API passes the following configuration parameters to the EMAC driver:

* EMAC port number
* EMAC packet receive filter level
* Multicast configurations

NOTE: This API is currently only implemented for v0 version of the driver.

### Queue Management

The EMAC driver manages one TX packet descriptor queue and one RX packet descriptor queue per each EMAC port, the TX/RX queue size is initialized by the application. The driver pre-allocates the packet buffer for each packet descriptor pushed to the RX queue when an EMAC port is opened. The driver frees both TX/RX queues when an EMAC port is closed.

### Packet Descriptor

By default, the EMAC driver uses CPPI RAM(8K-byte) for EMAC IP managed Packet Descriptor memory. This internal 8K-byte memory is used to manage the buffer descriptors that are 4-word(16-bytes) deep. The maximum number of descriptors that can be used for managing the packets being transferred is 512. Application shall allocate the packet descriptors for TX, RX and should pass the information to driver using EMAC\_OPEN\_CONFIG\_INFO\_T structure during driver open.

### Packet TX

The application can send a packet by calling *emac\_send* () API, the application needs to allocate an application managed packet descriptor from the application queue, copy the packet data and convert it to the EMAC driver managed packet descriptor format.

The following figure shows the EMAC/CSL API for a packet sent:-

Figure 3 : EMAC TX function

**EMAC**

**LLD**

**App**

emac\_send(port\_num, &app\_pkt\_desc)

**CSL**

EMAC\_sendPacket(hEmac, &csl\_pkt\_desc);

EMAC\_txEoiWrite(EMAC\_CORE\_NUM)\*

CSL\_semHwControl(hTxSem, SEM\_CMD\_FREE\_DIRECT, NULL)\*

EMAC\_TxServiceCheck(hEmac)

CSL\_semGetHwStatus(hTxSem, SEM\_QUERY\_DIRECT,&resp)\*

**Notes**

:

\* Only called for C6474 device

TX Interrupt

free\_pkt\_cb(port\_num, &app\_pkt\_desc)

### Packet RX

When a packet is received, the EMAC driver will convert the packet descriptor received to the application managed packet descriptor format and pass it to the application by calling the rx\_pkt\_cb() callback function.

The following figure shows the EMAC/CSL API for a packet received:-



Figure 4 : EMAC RX function

### Single Critical Section

The EMAC driver maintains certain per core specific data structures. These data structures need to be protected from access by multiple users running on the same core. Users are defined as entities in the system which uses the EMAC Driver API’s. The critical section defined here should also take into account the context of these users (Thread or Interrupt) and define the critical sections appropriately.

For example: In the EMAC RX interrupt service routine, if RX interrupt is not disabled, a new RX interrupt may pre-empt the existing RX ISR and cause data corruption in CSL CPPI packet descriptors.

The EMAC driver uses the Emac\_osalEnterSingleCoreCriticalSection() API to enter the single core critical section and Emac\_osalExitSingleCoreCriticalSection to exit the single core critical section.

### Multi-core Critical section

The EMAC driver supports multiple cores sharing the same EMAC port. The driver defines the following common data structures that are shared by all the cores:

* EMAC\_Device emac\_comm\_dev
* EMAC\_COMMON\_PCB\_T emac\_comm\_pcb

emac\_comm\_dev contains common EMAC device instance information, it is defined in the EMAC driver, but is managed by the EMAC CSL.

emac\_comm\_pcb contains common port control block information that is managed by the EMAC driver.

The EMAC driver defines a pragma data section “emacComm” for these two data structures, the application needs to put “emacComm” data section in the shared memory (either shared L2 data if available or external memory)

The EMAC driver calls Emac\_osalEnterMultipleCoreCriticalSection() and Emac\_osalExitMultipleCoreCriticalSection() API to enter and exit critical section to access shared resource by multiple cores. The EMAC multicore test application shows an example how to implement semaphore protection for shared resource access among multiple cores. C6472 uses IPC GateMP module to implement a software semaphore, and C6474 uses CSL hardware semaphore.

For shared memory access, the EMAC driver calls Emac\_osalBeginMemAccess() and Emac\_osalEndMemAccess() to protect cache coherence when cache is enabled. The driver always performs an invalidate cache operation before reading data and write back cache operation after writing data. The start address of emac\_comm\_dev and emac\_comm\_pcb need to be set aligned to the cache line size of the device by the application.

The following figure shows an example how the EMAC driver can access the shared resource:-



Figure 5 : EMAC Critical section access

### Interrupts

Interrupt configuration is specified in SOC’s init configuration and is provided to the EMAC LLD at time of *emac\_open* API. Interrupt registration is done within the LLD at time of *emac\_open*. Once interrupt is received, application provided callbacks are invoked.

## UDMA/NavSS based IP Driver: IP Version 5

IP version 5 supports SOCs based on NavSS/UDMA based DMA interface eg:AM65XX. The LLD provides a common set of APIs to service both CPSW and ICSS-G hardware IP ports. This is possible due to the NavSS IP which groups together various different hardware IP blocks ( in this case CPSW/ICSS-G) and whose purpose is to support the efficient transfer of data between various software, firmware and hardware entities via the use of channels.

A channel is a DMA instance/resource of which there are the following 2 types:

1. Receive (RX) Channel
   1. RX Packet Channel: EMAC LLD receives packets from the network via ICSSG/CPSW subsystem
   2. RX Management Channel EMAC LLD receives management packets (for example management request responses or TX timestamp or PSI) from ICSSG subsystem.

ICSS Switch supports 4 RX channels distributed as follows:

* Physical port 0  data  packets from network
* Management and related responses from firmware to EMAC LLD for Slice 0 (logical ‘half’ of an ICSS)
* Physical Port 1 data packets from network
* Management and related responses from firmware to EMAC LLD for Slice 1

Each RX channel can be “divided” into to N sub channels where each sub-channel can be considered a distinct flow having each having its own free and completion ring pair. This allow for “bining” packets of different types to be delivered to the EMAC LLD. ICSS-G f/w provides a flow id as packet meta data that the DMA uses to determine the ultimate free/completion ring pair to use. Note that when a channel is created using the UDMA driver,  a flow is created by “default” and is considered by the driver is the 1st sub-channel or flow. The remaining N-1 flows are sequential.

Default SOC configuration as specified by the emac\_soc.c file (see sub-sequent section for overview) provides configuration for the LLD to create  1 RX Packet Channel  with N sub-channels and 1 RX Management Response Channel with M sub-channels per slice.

1. Transmit (TX) Channel: EMAC LLD transmits packets to the network via ICSSG/CPSW subsystem

Default SOC configuration as specified in the emac\_soc.c file provides configuration for the LLD to create 4 TX channels per physical port (slice). LLD provides the application the option to choose which TX channel to transmit in the packet send function. More details about transmitting packets in subsequent sections below. Note the highest priority TX channel (i.e., channel 3) is used to carry  management messages to firmware as well.

### Memory

There is no constraint on where Packet Descriptor, packet buffer, and Ring memory resides except that the memory block/region must cache size aligned on cores which are not hardware cache coherent such as the R5F (there may be performance impacts however). Application shall allocate the packet descriptors for TX, RX and Ring memory and should pass the information to LLD using EMAC\_OPEN\_CONFIG\_INFO\_T structure during driver open. See section 7.4.2.1 for details.

### EMAC Driver Initialization configuration

The *emac\_open()* API is used for driver initialization. Refer to the EMAC\_OPEN\_CONFIG\_INFO\_T as defined in emac\_drv.h for details of configuration parameters passed into the driver at the time of *emac\_open* API call.

The EMAC driver unit test application provides a sample implementation sequence which initializes and configures the EMAC driver. This implementation is sample only and application developers are recommended to modify it as deemed fit.

The following flow diagram illustrates a sample calling sequence of API calls that can be used as a guideline for application development in addition to what is provided with the EMAC driver unit test application.



Figure 6 API Call Flow Sequence

#### EMAC Open Configuration Details.

EMAC\_OPEN\_CONFIG\_INFO\_T is passed into the driver at time of *emac\_open()* and is per port. The following table describe each of the fields of this structure.

|  |  |
| --- | --- |
| **Field** | **Description** |
| master\_core\_flag | Indicates core as master, used for multiple core use case and legacy SOCs. For Maxwell, only single core use case is currently supported, set this to TRUE. |
| mdio\_flag | Note that this does not apply for Maxwell, always set this to true required to poll for link status |
| phy\_addr | Note that this does not apply for Maxwell, phy address comes for SOC configuration mentioned in below table. |
| num\_of\_tx\_pkt\_desc | Total # of pkt desc initialized for all the TX chans in TX queues/rings |
| num\_of\_rx\_pkt\_desc | Total # of pkt desc initialized for all the RX chans in RX queues/rings |
| max\_pkt\_size | Maximum size of the packet in bytes |
| num\_of\_chans | Total number of TX/RX channels configured for this core, only applicable for SOC\_C6657,for Maxwell, channel configuration comes from SOC configuration mentioned in table below. |
| p\_chan\_mac\_addr | Note that this does not apply for Maxwell switch use case.  Use EMAC\_IOCTL\_INTERFACE\_MAC\_CONFIG |
| rx\_pkt\_cb | EMAC RX call back function to receive packets |
| alloc\_pkt\_cb | EMAC allocate packet call back function, used to setup RX channels/rings at startup and replenish rx free channel/ring at time of packet receive. |
| free\_pkt\_cb | EMAC free packet call back function, used to free packet packet to application during tx completion event, and time of driver close(at close for both rx and tx packets) |
| rx\_mgmt\_response\_cb | EMAC receive call back function for management response from ICSSG FW |
| tx\_ts\_cb | EMAC transmit timestamp response callback function |
| hwAttrs | Pointer to a driver specific hardware attributes structure, see below table for details |
| appPrivate | DEPRECATED: Application specific handle, not used by driver |
| mode\_of\_operation | DEPRECATED: should always be set to EMAC\_MODE\_INTERRUPT, to poll the driver for packets, use emac\_poll\_ctrl API. |
| udmaHandle | Handle to UDMA driver |
| drv\_trace\_cb | EMAC driver trace callback function |

##### SOC Specific Configuration

At time of *emac\_open*, the driver requires SOC specific configuration. This is passed to the driver via *hwAttrs* field of *EMAC\_OPEN\_CONFIG\_INFO\_T*. Defaults for soc configuration are in ti/drv/emac/soc/am65xx/emac\_soc.c. Since the emac drivers supports many SOC types, the hwAttrs field is passed in as a void \* and then internally casted by the driver to that specific SOC hw attributes structure. The hwAttrs struct defined for SOC\_AM65XX is EMAC\_HwAttrs\_V5

The following is EMAC\_HwAttrs\_V5 declaration:

typedef struct EMAC\_HwAttrs\_V5\_s {

/\*! Per Port configuration \*/

EMAC\_PER\_PORT\_CFG portCfg[EMAC\_MAX\_PORTS];

} EMAC\_HwAttrs\_V5;

This configuration consists of the following fields and is per port.

NOTE that unless stated, those fields should not be modified, fields which can be modified are in bold:

|  |  |
| --- | --- |
| **Field** | **Description** |
| phyAddr | PHY address (board specific) |
| nTxChans | number of transmit channels |
| mdioRegsBaseAddr | base address of MDIO sub-system |
| icssSharedRamBaseAddr | base address of ICSSG shared memory, N/A for CPSW2G |
| icssgCfgRegBaseAddr | base address of ICSSG configuration registers, N/A for CPSW2G |
| icssDram0BaseAddr | base address of ICSSG data ram 0 memory, N/A for CPSW2G |
| txChannel[4]  of type EMAC\_PER\_CHANNEL\_CFG\_TX | configuration required to setup up to 4 transmit channels which is to be provided by the application. This includes  **chHandle** :memory for Udma\_ChObj  **freeRingMem**: pointer to memory for free UDMA ring, sized to **elementCount** \* size of uint64\_t .  **compRingMem**: pointer to memory for UDMA completion ring, sized to number of ring entries \* size of uint64\_t .  NOTE: size should be the same as used for free ring mem.  **hPdMem**: pointer to memory for packet descriptors that will be pushed to the TX free descriptor software ring maintained by the driver. This needs to be sized by **num\_of\_tx\_pkt\_desc** (field of *EMAC\_OPEN\_CONFIG\_INFO\_T) \** 128*.*  **eventHandle**: placeholder for adding interrupt support at ring level, currently not used.  **elementCount**: Set to queue depth of the ring/number of ring elements (this can be modified, up to 128)  threadId: thread ID to connect cpdma to/from emac psi |
| rxChannel of type EMAC\_PER\_CHANNEL\_CFG\_RX | configuration required to setup upto a receive channel which is to be provided by the application. This includes  **chHandle** : pointer to memory for Udma\_ChObj  **flowHandle**: pointer to memory for Udma\_FlowObj to setup additional flows/rings in additional to default flow/ring.  nsubChan: number of sub-channels or flows.  subChan: sub-channel specific configuration as specified by  EMAC\_RX\_SUBCHAN which has the following fields:  nfreeRings: number of free rings associated with sub-channel, for now set to 1 but can be upto4 to support advanced features of flow.  Note: EMAC\_MAX\_FREE\_RINGS\_PER\_SUBCHAN is set to 1.  **freeRingHandle**[EMAC\_MAX\_FREE\_RINGS\_PER\_SUBCHAN]: pointer to memory for free UDMA ring handle  **freeRingMem**[EMAC\_MAX\_FREE\_RINGS\_PER\_SUBCHAN]: pointer to memory for free UDMA ring, sized to **elementCount** \* size of uint64\_t .  **compRingHandle**: pointer to memory for free UDMA completion handle  **compRingMem**: pointer to memory for UDMA completion ring, sized to number of ring entries \* size of uint64\_t ..  **hPdMem**[EMAC\_MAX\_FREE\_RINGS\_PER\_SUBCHAN]: memory for packet descriptors that will be pushed to the rx free ring. This needs to be sized by **num\_of\_rx\_pkt\_desc** (field of *EMAC\_OPEN\_CONFIG\_INFO\_T) \** 128*.*  **eventHandle**: placeholder for adding interrupt support at ring level, currently not used.  **elementCountFree**[EMAC\_MAX\_FREE\_RINGS\_PER\_SUBCHAN]: set to queue depth of the rx free ring  **elementCountCompletion**: set to queue depth of the rx completion ring  threadId: thread ID to connect cpdma to/from emac psi |
| rxChannelCfgOverPSI | See details above, this channel is used to provide management responses from FW. |
| rxChannel2CfgOverPSI | See details above, this channel is used to provide transmit timestamp responses from FW. |
| *getFwCfg* | Function pointer provided by the application and used by the driver to get firmware specific configuration, required to setup the environment for FW to operate. Must be set to *emacGetSwitchFwConfig* or switch use case and *emacGetDualMacFwConfig* for dual mac use case. |

#### ICSSG Port Queue Configuration

The firmware configuration is directly tied to the ICSSG firmware and must not be modified accept for addresses of application provided memory for ICSSG port queues. This memory must be allocated from MSMC SRAM and128 byte aligned. The size of each port queue per ICSSG instance is 144000 bytes for switch and 100352 bytes for dual mac.

The following sequence is an example illustration of how to accomplish this (NOTE the ports used in this illustration are software logical port 0-4, not EMAC\_SWITCH\_PORT1/ EMAC\_SWITCH\_PORT2 and also shown only for ICSSG\_0.

/\* memory of port queue, from msmc memory, 128 byte aligned \*/

uint8\_t icss\_tx\_port\_queue\_icssg0[1][144000] \_\_attribute\_\_ ((aligned (UDMA\_CACHELINE\_ALIGNMENT))) \_\_attribute\_\_ ((section (".bss:emac\_msmc\_mem")));

EMAC\_FW\_APP\_CONFIG \*pFwAppCfg;

/\* Get application part of init config \*/

*emacGetSwitchFwAppInitCfg(port\_num, &pFwAppCfg);*

/\* update the address lo/hi with address of port queue \*/

pFwAppCfg->txPortQueueLowAddr = 0xFFFFFFFF & ((uint32\_t) &icss\_tx\_port\_queue\_icssg0[0][0]);

pFwAppCfg->txPortQueueHighAddr = 0;

*/\* set/store the addresses of the port queue\*/*

*emacSetSwitchFwAppInitCfg(port\_num, pFwAppCfg);*

### Packet TX

EMAC LLD for AM65XX will support UDMAP operations to transfer data between the host processor and network peripherals. A valid port number and EMAC\_PKT\_DESC\_T are required arguments to the *emac\_send* API.

For DUAL MAC use case, the port number is the physical port used to transmit the packet to the network.

For switch use case, we will use virtual port concept for the port number when calling *emac\_send*. For directed packet to a specific physical port, use [virtual] port EMAC\_SWITCH\_PORT1 to send on physical port 0 of the switch and use [virtual] port EMAC\_SWITCH\_PORT2 to send on physical port 1 of the switch. For un-directed packets use virtual port EMAC\_SWITCH\_PORT. In this case, the driver will take care of transmitting the packet out on the switch port(s).

The transmit submit ring to be used for the transmission should be specified in the PktChannel field of the EMAC\_PKT\_DESC\_T passed in. In other words, the application decides on which of the Transmit Channels(rings) to use. The driver will support configuration of up to 4 TX channels per port (at time of *emac\_open*) each associated with a transmit submit ring/completion ring pair.

The TX port queue (0-7) inside ICSSG that is used to transmit the packet from the ICSSG firmware to the PHY can be specified in the *TxPktTc* field of the EMAC\_PKT\_DESC\_T passed in in the *emac\_send* API call. In other words, the application can select which TX port queue to use.

*Future Development: The application can set the TxPktTc field to 0xFF and in this case, the firmware will determine which port queue to use based on the priority REGEN(remap) and PORT PRIORITY mapping that is configured for the host port.*

The following sequence occurs during *emac\_send* API call:

1. LLD/driver maintains hardware TX descriptor free linked list (in software) setup at time of *emac\_open*
2. At time of *emac\_send* API call, LLD will pop a free TX descriptor from free linked list and populate free TX descriptor with packet length, pointer to packet buffer and any META data that is passed in the application descriptor. This provides ZERO copy transfer of data. ZERO copy is achieved by transferring ownership of the passed in descriptor and linked packet buffer to the LLD which is directly “linked” to the TX descriptor which is queued on the Transmit Submit ring. Note that no “memcpy” is performed by the driver during *emac\_send* API call.
3. LLD will push the TX descriptor using UDMA ring queue API to the specified transmit submit ring associated with the port number passed into the API call.
4. The *emac\_send* will return failure codes to the calling application based on the failure encountered as follows:
   1. Port is closed : EMAC\_DRV\_RESULT\_SEND\_ERR
   2. Unable to submit the packet for transmission on the transmit ring due to the ring being full (in this case the buffer ownership is returned to the application): EMAC\_DRV\_RESULT\_ERR\_UDMA\_RING\_ENQUEUE
   3. No free TX descriptor available: EMAC\_DRV\_RESULT\_ERR\_NO\_FREE\_DESC
   4. Invalid packet channel specified in EMAC\_PKT\_DESC: EMAC\_DRV\_RESULT\_ERR\_INVALID\_CHANNEL

NOTE: The DMA is used to move the packet from the host owned buffer to ICSSG firmware, which then will copy the packet to the TX port queue. Thus the DMA for TX packets in TX ring is controlled by firmware and firmware will not allow the DMA of the packet if there is no room in TX port queue in ICSS.  So packets will remain in the TX ring in this event and firmware will service another TX channel.

### TX Software Descriptor Return Queue Processing

Note that at the time of *emac\_send()*, the software descriptor passed in’s ownership is given to the driver and needs to returned back to the calling application as specified in the section above. The following mean is provided by the driver to provide the software descriptor back to the calling application.

1. *emac\_poll\_ctrl*() API will directly query the TX completion queue and invoke the TX callback if packet is present in completion queue. *emac\_poll\_ctrl*() call will return to the user after packets in the desired rings are serviced. When

### TX Time Stamp

The *emac\_send()* API will allow a packet to be marked as TX timestamp required and will allow the application to provide a piece of opaque data (i.e. timestamp id) that can be used to associate a TX timestamp with the packet when the TX timestamp is delivered later. Application will need to register a callback at time of *emac\_open* which driver will call to provide the timestamp. The following is the prototype of the callback:

typedef void EMAC\_TX\_TS\_CALLBACK\_FN\_T

(

uint32\_t port\_num,

/\*\*< EMAC port number \*/

uint32\_t ts\_id,

/\*\*< timestamp id to correlate TS response with TX request \*/

uint64\_t ts

/\*\*< 64 bit timestamp provided by ICSSG FW \*/

bool isValid;

/\*\*< flag to indicate if packet was transmitted and timestamp is valid \*/

);

The callback to the application to free the TX descriptor is completely independent of the callback to the application for TX timestamp response. There is no sycnronization between when the TX packet descriptor is returned to the application via the packet free callback and when the TX Timestamp response callback is issued.

The application should not re-use a TX packet descriptor which has a pending TX timestamp request until it receives the TX timestamp response even though the TX packet descriptor has been returned to the application via the packet free callback.

.

In order to request the TX timestamp, the application will need to update the following fields of the EMAC\_PKT\_DESC\_T when calling the *emac\_send* API:

1. Update Flags field in EMAC\_PKT\_DESC\_T with EMAC\_PKT\_FLAG\_TX\_TS\_REQ
2. Update ts\_id field in EMAC\_PKT\_FLAG\_TX\_TS\_REQ with 32 bit id which will be returned with timestamp and can be used by application to correlate TX timestamp request with response

In order to retrieve the timestamp, the application will need to use the *emac\_poll\_ctrl* API as follows:

1. emac\_poll\_ctrl(port\_num, rxPktRings, rxMgmtRings, txRings) where rxMgmtRings is a bitmap of RX Management rings to poll and needs to have bit 2 set. If timestamp management packet is available, registered TX timestamp callback will be invoked.

### Packet RX

The driver supports multiple receive rings per port. In the case of ICSSG Switch, 9 receive rings are supported. Currently, only 8 of the receive rings are used and packets are directed to the rings based on the PCP to port queue mapping as specified by the following IOCTL: EMAC\_IOCTL\_PORT\_PRIO\_MAPPING\_CTRL. For ICSSG Dual MAC, 1 receive ring is currently supported.

When a packet is received, the EMAC driver will convert the packet descriptor received to the application managed packet descriptor format (EMAC\_PKT\_DESC\_T) and pass it to the application by calling the rx\_pkt\_cb() callback function. The receive ring the packet arrives on will be updated in the PktChannel field of the application managed packet descriptor being returned via the callback.

Registration of rx\_pkt\_cb() is done at time of *emac\_open()* API call. For both mode of operation specified below, rx\_pkt\_cb() will get called to provide the packet to the application.

For receive packets, the following 2 modes of operation are supported and can be configured at time of *emac\_open()* for specified port (note the default mode is INTERRUPT).

1. EMAC\_MODE\_INTERRUPT: *emac\_poll\_pkt*() API will PEND on a SEMAPHORE which is posted by RX ISR(ISR registration/SEMAPHORE creation done at time of *emac\_open*),  invoke RX callback and again PEND on SEMAHPORE. This is a blocking API call and will only return to user   application if PORT status is closed. Task context is required in INTERRUPT mode.
2. EMAC\_MODE\_POLL: *emac\_poll\_ctrl*() API will directly query the RX completion queue and invoke the RX callback if packet is received. Will return to user application after each *emac\_poll\_ctrl*() call.

### RX Time Stamp

The 64 bit RX timestamp can be extracted from the psinfo[] field of the EMAC\_CPPI\_DESC\_T hardware descriptor which is dequeued from the UDMA ring. psinfo[1] contains the upper 32 bits of timestamp and psinfo[0] contains lower 32 bits of timestamp.

The RxTimeStamp field of the EMAC\_PKT\_DESC\_T will be updated with the 64 bit timestamp from the hardware descriptor and provided for each packet received and provided to the calling application via RX callback.

### New APIs

API to retrieve ICSS-G hardware statistics (NOTE that some of the hardware stats saturates much faster due to 16-bit counter in ICSS-G hardware):

1. EMAC\_DRV\_ERR\_E *emac\_get\_statistics\_icssg*(uint32\_t port\_num, EMAC\_STATISTICS\_ICSS-G\_T \*p\_stats, bool clear)

API to poll receive packet rings, receive management response rings and tx completion rings. Note that a task context is required to make this call and this function does not return until all of the rings per the configured bitmaps have been examined.

1. EMAC\_DRV\_ERR\_E emac\_poll\_ctrl(uint32\_t port\_num, uint32\_t rxPktRings, uint32\_t rxMgmtRings, uint32\_t txRings)

rxPktRings is a bitmap of which packet completion rings to poll. Please refer to EMAC\_POLL\_RX\_PKT\_RINGS enum for configuration values. To poll multiple rings, these enum values can be “orred” together. First ring (EMAC\_POLL\_RX\_PKT\_RING1) is not used by firmware, The assignment of packets to rings EMAC\_POLL\_RX\_PKT\_RING2 - EMAC\_POLL\_RX\_PKT\_RING9 is based on EMAC\_IOCTL\_PORT\_PRIO\_MAPPING\_CTRL configuration.

rxgmtRings is a bitmap of which management rings to poll. Please refer to EMAC\_POLL\_RX\_MGMT\_RINGS enum for configuration values. To poll multiple rings, these enum values can be “orred” together. First ring (EMAC\_POLL\_RX\_MGMT\_RING1) is not used by firmware, to receive mgmt response from FW, use EMAC\_POLL\_RX\_MGMT\_RING2 and to receive transmit timestamp response from FW, use EMAC\_POLL\_RX\_MGMT\_RING3.

txRings is a bitmap of which packet transmit completion rings to poll. Please refer to EMAC\_POLL\_TX\_COMPL\_RINGS enum for configuration values. To poll multiple rings, these enum values can be “orred” together. EMAC\_POLL\_TX\_COMPLETION\_RING1 is the lowest priority TX channel and EMAC\_POLL\_TX\_COMPLETION\_RING4 is the highest priority TX channel.

API for issuing IOCTL commands for ICSSG ports

1. EMAC\_DRV\_ERR\_E *emac\_ioctl*(uint32\_t port\_num, EMAC\_IOCTL\_CMD emacIoctlCmd, EMAC\_IOCTL\_PARAMS \*emacIoctlParams)

Callback API for receiving IOCTL command response from FW

1. EMAC\_RX\_MGMT\_CALLBACK\_FN\_T

typedef void EMAC\_RX\_MGMT\_CALLBACK\_FN\_T

(

uint32\_t port\_num,

/\*\*< EMAC port number \*/

EMAC\_IOCTL\_CMD\_RESP\_T\* pCmdResp

/\*\*< Pointer to the IOCTL command reponse \*/

);

Application will need to register this callback function with driver at time of *emac\_open()* by populating rx\_mgmt\_response\_cb of EMAC\_OPEN\_CONFIG\_INFO\_T. The

### IOCTL API Details

IOCTL can be classified into the following 2 types:

1. Application invokes IOCTL which results in driver issuing MMR update, non-blocking where IOCTL is synchronous in nature and executes immediately with return code status.
2. Application invokes an IOCTL which results in driver issuing management (MGMT) message over PSIL interface to firmware (FW) over TX channel, this does not complete until MGMT response is returned to calling application via RX management channel. This is an asynchronous call. As part of the IOCTL, a sequence number is used for each IOCTL which is returned to the calling application to correlate an IOCTL request with a response. NOTE: At any given time, only 1 IOCTL request can be outstanding. If 1 IOCTL request is in progress and application issues a 2nd one, it will get rejected with error code EMAC\_DRV\_RESULT\_IOCTL\_ERR. If the driver is able to issue the IOCTL request to the FW, it will return EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS.

The result of the IOCTL call is made available to the calling application with EMAC\_RX\_MGMT\_CALLBACK\_FN\_T. This is a callback function the application is required to register with the driver at time of *emac\_open().* Refer to the *status* field of EMAC\_IOCTL\_CMD\_RESP\_T

typedef void EMAC\_RX\_MGMT\_CALLBACK\_FN\_T

(

uint32\_t port\_num,

/\*\*< EMAC port number \*/

EMAC\_IOCTL\_CMD\_RESP\_T\* pCmdResp

/\*\*< Pointer to the IOCTL command reponse \*/

);

NOTE: IOCTL tables below will indicate IOCTL type to be either synchronous or asynchronous, also provide details of the *status* field of EMAC\_IOCTL\_CMD\_RESP\_T.

Refer to the following ladder diagram for details of asynchronous IOCTL call flow.



Figure 7 Asynchronous IOCTL Call Flow

The tables below provide a list of IOCTLS currently supported for switch and dual mac use case.

#### Switch use case

The following virtual ports should be used when making IOCTL calls as indicated in the table below:

#define EMAC\_SWITCH\_PORT0 ((uint32\_t)9U) 🡺 Host Port

#define EMAC\_SWITCH\_PORT1 ((uint32\_t)10U) 🡺 ETH0/SW0 configuration

#define EMAC\_SWITCH\_PORT2 ((uint32\_t)11U) 🡺 ETH1/SW1 configuration

#define EMAC\_SWITCH\_PORT ((uint32\_t)12U) 🡺 Switch centric configuration

|  |  |  |  |
| --- | --- | --- | --- |
| IOCTL Command/Sub Command | Description | IOCTL Parameters | RETURN TYPE |
| EMAC\_IOCTL\_FDB\_ENTRY\_CTRL/  EMAC\_IOCTL\_FDB\_ENTRY\_ADD  Type: asynchronous IOCTL  MGMT message over PSI to FW, MGMT response over PSI from FW via RX MGMT channel.  NOTE: To program FDB entry as a special management multicast frame set the block and secure bits in FID\_C2. Please refer FID\_C2 Bitfield in AM654x\_PROFINET\_Switch\_Non\_Real\_time\_Interface\_Design pdf.  fdbEntry field (also the FID\_C2) is now an array of 2 and it’s possible to assign two values to the two physical ports. Index 0 is for EMAC\_SWITCH\_PORT1 and Index 1 is for EMAC\_SWITCH\_PORT2. | Add forward data base entry to internal ICSSG memory. | **PORT\_NUM:**  EMAC\_SWITCH\_PORT  **EMAC\_IOCTL\_FDB\_ENTRY:**  uint8\_t mac[6]  int16\_t vlanId  vlanId Range: 0 to 4095  uint8\_t fdbEntry[2] | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_INVALID\_VLAN\_ID on vlanId out of range.  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT message over PSI as a result of UDMA ring enqueue failure or free TX descriptor not available  Status field values returned via EMAC\_RX\_MGMT\_CALLBACK\_FN\_T :  0x1: returned on success.  0x3: returned if an ageable FDB entry is removed in order to ADD the new entry. Aged out entry will be returned to the in the 1st 2 bytes of the *respParams* field of the EMAC\_IOCTL\_CMD\_RESP\_T.  0x10: returned on error (no free entry available) |
| EMAC\_IOCTL\_FDB\_ENTRY\_CTRL/ EMAC\_IOCTL\_FDB\_ENTRY\_DEL  Type: asynchronous IOCTL  MGMT message over PSI to FW, MGMT response over PSI from FW via RX MGMT channel | Delete forward data base entry from internal ICSSG memory | **PORT\_NUM:**  EMAC\_SWITCH\_PORT  **EMAC\_IOCTL\_FDB\_ENTRY:**  uint8\_t mac[6]  int16\_t vlanId  vlanId Range: 0 to 4095  uint8\_t fdbEntry  Note: No need to populate fdb\_entry field | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_INVALID\_VLAN\_ID on vlanId out of range.  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT msg over PSI as a result of UDMA ring enqueue failure or TX descriptor not available.  Status field values returned via EMAC\_RX\_MGMT\_CALLBACK\_FN\_T :  0x1: returned on success.  0x10: returned on error (entry to delete not found) |
| EMAC\_IOCTL\_FDB\_ENTRY\_CTRL/ EMAC\_IOCTL\_FDB\_ENTRY\_DELETE\_ALL or EMAC\_IOCTL\_FDB\_ENTRY\_AGEABLE  Type: asynchronous IOCTL  MGMT message over PSI to FW, MGMT response over PSI from FW via RX MGMT channel.  **For future deliverable, subject to change.** | Delete all forward data base entries from internal ICSSG memory | **PORT\_NUM:**  EMAC\_SWITCH\_PORT | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT msg over PSI as a result of UDMA ring enqueue failure or TX descriptor not available.  Status field values returned via EMAC\_RX\_MGMT\_CALLBACK\_FN\_T :  0x1: returned on success.  0x10: returned on error |
| EMAC\_IOCTL\_PORT\_STATE\_CTRL/ EMAC\_IOCTL\_PORT\_STATE\_DISABLE  Type: asynchronous IOCTL  MGMT message over PSI to FW, MGMT response over PSI from FW via RX MGMT channel. | Place PORT is disabled state | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1 or EMAC\_SWITCH\_PORT2 | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT msg over PSI as a result of UDMA ring enqueue failure or TX descriptor not available.  Status field values returned via EMAC\_RX\_MGMT\_CALLBACK\_FN\_T :  0x1: returned on success. |
| EMAC\_IOCTL\_PORT\_STATE\_CTRL/ EMAC\_IOCTL\_PORT\_STATE\_BLOCKING  Type: asynchronous IOCTL  MGMT message over PSI to FW, MGMT response over PSI from FW via RX MGMT channel. | Place PORT is blocking state | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1 or EMAC\_SWITCH\_PORT2 | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT msg over PSI as a result of UDMA ring enqueue failure or TX descriptor not available.  Status field values returned via EMAC\_RX\_MGMT\_CALLBACK\_FN\_T :  0x1: returned on success. |
| EMAC\_IOCTL\_PORT\_STATE\_CTRL/ EMAC\_IOCTL\_PORT\_STATE\_FORWARD  Type: asynchronous IOCTL  MGMT message over PSI to FW, MGMT response over PSI from FW via RX MGMT channel. | Place PORT is forwarding state | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1 or EMAC\_SWITCH\_PORT2 | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT msg over PSI as a result of UDMA ring enqueue failure or TX descriptor not available.  Status field values returned via EMAC\_RX\_MGMT\_CALLBACK\_FN\_T :  0x1: returned on success. |
| EMAC\_IOCTL\_PORT\_STATE\_CTRL/ EMAC\_IOCTL\_PORT\_STATE\_FORWARD\_WO\_LEARNING  Type: asynchronous IOCTL  MGMT message over PSI to FW, MGMT response over PSI from FW via RX MGMT channel. | Place PORT is forwarding state without learning | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1 or EMAC\_SWITCH\_PORT2 | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT msg over PSI as a result of UDMA ring enqueue failure or TX descriptor not available.  Status field values returned via EMAC\_RX\_MGMT\_CALLBACK\_FN\_T :  0x1: returned on success. |
| EMAC\_IOCTL\_VLAN\_CTRL/ EMAC\_IOCTL\_VLAN\_SET\_DEFAULT\_TBL  Type: synchronous IOCTL  MMR update, non-blocking | Update ICSSG shared memory with default vlan fid table entries (4096 entries set to default settings) | **PORT\_NUM:**  EMAC\_SWITCH\_PORT | EMAC\_DRV\_RESULT\_OK on success |
| EMAC\_IOCTL\_VLAN\_CTRL/ EMAC\_IOCTL\_VLAN\_SET\_ENTRY  Type: synchronous IOCTL  MMR update, non-blocking | Set entry in vlan table for specified vlan id value where vlan id is from 0 to 4095). | **PORT\_NUM:**  EMAC\_SWITCH\_PORT  **EMAC\_IOCTL\_VLAN\_FID\_ENTRY**  int16\_vlanId  vlanId Range: 0 to 4095  EMAC\_IOCTL\_VLAN\_FID\_PARAMS vlanFidPrams –used to populate vlan\_fid and vlan\_info | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_INVALID\_VLAN\_ID on failure |
| EMAC\_IOCTL\_VLAN\_CTRL/ EMAC\_IOCTL\_VLAN\_SET\_DEFAULT\_VLAN\_ID  Type: synchronous IOCTL  MMR update, non-blocking | Set default VLAN ID and PCP bits for specified switch port. | **PORT\_NUM:**  EMAC\_SWITCH\_PORT0(host port)  or  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2  **EMAC\_IOCTL\_VLAN\_DEFAULT\_ENTRY**  int16\_vlanId  vlanId Range: 0 to 4095  int8\_t pcp  range 0-7 | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_INVALID\_VLAN\_ID on failure |
| EMAC\_IOCTL\_VLAN\_CTRL/ EMAC\_IOCTL\_VLAN\_GET\_ENTRY  Type: synchronous IOCTL  MMR update, non-blocking | Get entry in vlan table for specified vlan id value where vlan id is from 0 to 4095). | **PORT\_NUM:**  EMAC\_SWITCH\_PORT  **EMAC\_IOCTL\_VLAN\_FID\_ENTRY** to be populated by the driver. | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_INVALID\_VLAN\_ID on failure |
| EMAC\_IOCTL\_VLAN\_CTRL/ EMAC\_IOCTL\_VLAN\_AWARE\_MODE  Type: asynchronous IOCTL  MMR update and MGMT message over PSI to FW, MGMT response over PSI from FW via RX MGMT channel.  **For future deliverable, subject to change.** | Enable/disable VLAN aware mode. | **PORT\_NUM:**  EMAC\_SWITCH\_PORT,  enable(1), disable(0) | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR on failure |
| EMAC\_IOCTL\_PRIO\_REGEN\_CTRL  Type: synchronous IOCTL  MMR update, non-blocking | Configure the priority regeneration table for a port including the host port. | **PORT\_NUM:** EMAC\_SWITCH\_PORT1 OR EMAC\_SWITCH\_PORT2  **EMAC\_IOCTL\_PRIO\_REGEN\_MAP:**  8-byte priority regen array indexed by PCP value. Index 0 of the array corresponds to PCP 0 so if you want to change PCP 0 to 7 then you would write a value of 7 at index 0 | EMAC\_DRV\_RESULT\_OK on success |
| EMAC\_IOCTL\_PORT\_PRIO\_MAPPING\_CTRL  Type: synchronous IOCTL  MMR update, non-blocking | Configure the mapping of PCP to port queue. Also configures the mapping of PCP to CPPI flow/host receive ring. | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1 or EMAC\_SWITCH\_PORT2  **EMAC\_IOCTL\_PORT\_PRIO\_MAP:**  8-byte port priority to port queue mapping indexed by PCP value.  Also the 8-byte priority to CPPI flow/receive ring.    One-to-one mapping from PCP to output Queue is managed using FT3[0:7] and Classifier[0:7] MMRs. | EMAC\_DRV\_RESULT\_OK on success |
| EMAC\_IOCTL\_ACCEPTABLE\_FRAME\_CHECK\_CTRL/EMAC\_IOCTL\_ACCEPTABLE\_FRAME\_CHECK\_ONLY\_VLAN\_TAGGED  Type: asynchronous IOCTL  MGMT message over PSI to FW, MGMT response over PSI from FW via RX MGMT channel. | Admit only VLAN-tagged frames | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1 or EMAC\_SWITCH\_PORT2 | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT msg over PSI as a result of UDMA ring enqueue failure or TX descriptor not available.  Status field values returned via EMAC\_RX\_MGMT\_CALLBACK\_FN\_T :  0x1: returned on success. |
| EMAC\_IOCTL\_ACCEPTABLE\_FRAME\_CHECK\_CTRL/ EMAC\_IOCTL\_ACCEPTABLE\_FRAME\_CHECK\_ONLY\_UN\_TAGGED\_PRIO\_TAGGED  Type: asynchronous IOCTL  MGMT message over PSI to FW, MGMT response over PSI from FW via RX MGMT channel. | Admit Only Untagged and Priority-tagged frames | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1 or EMAC\_SWITCH\_PORT2 | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT msg over PSI as a result of UDMA ring enqueue failure or TX descriptor not available.  Status field values returned via EMAC\_RX\_MGMT\_CALLBACK\_FN\_T :  0x1: returned on success. |
| EMAC\_IOCTL\_ACCEPTABLE\_FRAME\_CHECK\_CTRL/ EMAC\_IOCTL\_ACCEPTABLE\_FRAME\_CHECK\_ALL\_FRAMES  Type: asynchronous IOCTL  MGMT message over PSI to FW, MGMT response over PSI from FW via RX MGMT channel | Admit all frames (default setting) | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1 or EMAC\_SWITCH\_PORT2 | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT msg over PSI as a result of UDMA ring enqueue failure or TX descriptor not available.  Status field values returned via EMAC\_RX\_MGMT\_CALLBACK\_FN\_T :  0x1: returned on success. |
| EMAC\_IOCTL\_INTERFACE\_MAC\_CONFIG  Type: synchronous IOCTL  MMR update, non-blocking | Interface MAC address configuration in hardware MMR’s. | **PORT\_NUM:**  EMAC\_SWITCH\_PORT0 (host port)  or  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2  **EMAC\_MAC\_ADDR\_T:**  uint8\_t   addr[EMAC\_MAC\_ADDR\_LENTH];  where EMAC\_MAC\_ADDR\_LENTH is 6 | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
| EMAC\_IOCTL\_SAV\_CHECK\_CTRL  Type: asynchronous IOCTL  MGMT message over PSI to firmware, MGMT response over PSI via RX MGMT channel.  **For future deliverable, subject to change.** | Source address violation check enable/disable control. | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1 or EMAC\_SWITCH\_PORT2  enable(1), disable(0) | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR on failure |
| EMAC\_IOCTL\_CUT\_THROUGH\_CTRL  Could use a classi to determine if a packet type is eligible for cut-through. Could also be 1 byte in SMEM per port which FW reads to determine if a packet type is eligible for cut-through  **For future deliverable, subject to change.** | TBD | TBD | TBD |
| EMAC\_IOCTL\_HOST\_TX\_RATE\_LIMITER\_CTRL  **For future deliverable, subject to change.** | TBD | TBD | TBD |
| EMAC\_IOCTL\_HOST\_RX\_RATE\_LIMTER\_CTLR  **For future deliverable, subject to change.** | TBD | TBD | TBD |
| EMAC\_IOCTL\_PKT\_TO\_FLOW\_CLASSI\_CTRL  **For future deliverable, subject to change.** | TBD | TBD | TBD |
| EMAC\_IOCTL\_UC\_FLOODING\_CTRL/ EMAC\_IOCTL\_PORT\_UC\_FLOODING\_ENABLE  Type: asynchronous IOCTL  MGMT message over PSI to firmware, MGMT response over PSI via RX MGMT channel. | Enable flooding of unknown unicast packets to host port | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2 | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT message over PSI as a result of UDMA ring enqueue failure or free TX descriptor not available  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
| EMAC\_IOCTL\_UC\_FLOODING\_CTRL/ EMAC\_IOCTL\_PORT\_UC\_FLOODING\_DISABLE  Type: asynchronous IOCTL  MGMT message over PSI to firmware, MGMT response over PSI via RX MGMT channel. | Disable flooding of unknown unicast packets to host port | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2 | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT message over PSI as a result of UDMA ring enqueue failure or free TX descriptor not available  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
| EMAC\_IOCTL\_FRAME\_PREEMPTION\_CTRL/ EMAC\_IOCTL\_PREEMPT\_TX\_ENABLE  Type: asynchronous IOCTL  MGMT message over PSI to firmware, MGMT response over PSI via RX MGMT channel. | Enable pre-emption on TX | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2 | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT message over PSI as a result of UDMA ring enqueue failure or free TX descriptor not available  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
| EMAC\_IOCTL\_FRAME\_PREEMPTION\_CTRL/ EMAC\_IOCTL\_PREEMPT\_TX\_DISABLE  Type: asynchronous IOCTL  MGMT message over PSI to firmware, MGMT response over PSI via RX MGMT channel. | Disable pre-emption on TX | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2 | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG if unable to send MGMT message over PSI as a result of UDMA ring enqueue failure or free TX descriptor not available  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
| EMAC\_IOCTL\_FRAME\_PREEMPTION\_CTRL/ EMAC\_IOCTL\_PREEMPT\_GET\_TX\_ENABLE\_STATUS  Type: synchronous IOCTL, non-blocking | Get status of pre-emption on TX | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2  **EMAC\_IOCTL\_PREEMPTION\_ENTRY: premt\_tx\_enabled\_status** field will be populated by the driver as follows **:** 1 if active, 0 if not active | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
| EMAC\_IOCTL\_FRAME\_PREEMPTION\_CTRL/ EMAC\_IOCTL\_PREEMPT\_GET\_TX\_ACTIVE\_STATUS  Type: synchronous IOCTL, non-blocking | Get status of weather pre-emption is active | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2  **EMAC\_IOCTL\_PREEMPTION\_ENTRY: premt\_tx\_active\_status** field will be populated by the driver as follows **:** 1 if active, 0 if not active | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
| EMAC\_IOCTL\_FRAME\_PREEMPTION\_CTRL/ EMAC\_IOCTL\_PREEMPT\_VERIFY\_ENABLE  Type: synchronous IOCTL  MMR update, non-blocking | Enable verify state machine | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2 | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
| EMAC\_IOCTL\_FRAME\_PREEMPTION\_CTRL/ EMAC\_IOCTL\_PREEMPT\_VERIFY\_DISABLE  Type: synchronous IOCTL  MMR update, non-blocking | Disable verify state machine | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2 | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
| EMAC\_IOCTL\_FRAME\_PREEMPTION\_CTRL/ EMAC\_IOCTL\_PREEMPT\_GET\_VERIFY\_STATE  Type: synchronous IOCTL, non-blocking | Get the verify state machine current state | EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2  **EMAC\_IOCTL\_PREEMPTION\_ENTRY: preempt\_verify\_state** enum field will be populated by the driver as follows:  STATE\_UNKNOWN  STATE\_INITIAL  STATE\_VERIFYING  STATE\_SUCCEEDED  STATE FAILED  STATE DISABLED | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
| EMAC\_IOCTL\_FRAME\_PREEMPTION\_CTRL/ EMAC\_IOCTL\_PREEMPT\_GET\_MIN\_FRAG\_SIZE\_LOCAL  Type: synchronous IOCTL, non-blocking | Get minimum fragment size supported by firmware | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2  **EMAC\_IOCTL\_PREEMPTION\_ENTRY: premt\_min\_fragment\_size** field will be populated with min fagment size supported | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
| EMAC\_IOCTL\_FRAME\_PREEMPTION\_CTRL/ EMAC\_IOCTL\_PREEMPT\_SET\_MIN\_FRAG\_SIZE\_REMOTE  Type: synchronous IOCTL  MMR update, non-blocking | Configure the minimum non final fragment size supported by remote link partner in units of 64 | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2  **EMAC\_IOCTL\_PREEMPTION\_ENTRY: premt\_min\_fragment\_size** field will be populated with min fagment size supported | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
| EMAC\_IOCTL\_CUT\_THROUGH\_PREEMPT\_SELECT  Type: synchronous IOCTL  MMR update, non-blocking | Configures queues are pre-emptive/express and/or as cut-through/Store&Forward | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2  **EMAC\_IOCTL\_PREMPT\_OR\_CUT\_THROUGH\_MAP –** The struct takes two arrays as inputs pcpPreemptMap is a byte map for each queue where 1 indicates that the queue is a pre-emptive queue. (This is not operational right now)  pcpCutThroughMap is similar but for determining if a queue should be cut-through or not | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
| EMAC\_IOCTL\_SPECIAL\_FRAME\_PRIO\_CONFIG  Type: synchronous IOCTL  MMR update, non-blocking | Specifies thequeue number to be used for special packets | **PORT\_NUM:**  EMAC\_SWITCH\_PORT1  or EMAC\_SWITCH\_PORT2  **EMAC\_IOCTL\_SPECIAL\_FRAME\_DEFAULT\_PRIO –** specifies thequeue number to be used for special packets | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_INVALID\_PORT on failure |
|  |  |  |  |

Table 2 Switch IOCTL Commands

#### DUAL MAC Use Case (Single Instance ICSS FW)

For DUAL MAC use case, use the software port number in the IOCTL call as follows:

ICSSG0 port 0, use LLD 0

ICSSG0 port 1, use LLD 1

ICSSG1 port 0, use LLD 2

ICSSG1 port 1, use LLD 3

ICSSG2 port 0, use LLD 4

ICSSG2 port 1, use LLD 5

|  |  |  |  |
| --- | --- | --- | --- |
| IOCTL Command/Sub Command | Description | IOCTL Parameters | RETURN TYPE |
| EMAC\_IOCTL\_PROMISCOUS\_MODE\_CTRL  (MMR update, non-blocking)  Type: synchronous IOCTL  MMR update, non-blocking | Enable/disable promiscuous mode of operation | Port number  enable(1), disable(0) | EMAC\_DRV\_RESULT\_OK on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR on failure |
| EMAC\_IOCTL\_PORT\_CTRL/ EMAC\_IOCTL\_PORT\_STATE\_DISABLE  **For future deliverable, subject to change.** | Place PORT is disabled state | port number | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR on failure |
| EMAC\_IOCTL\_PORT\_CTRL/ EMAC\_IOCTL\_PORT\_STATE\_ENABLE  **For future deliverable, subject to change.** | Re-enable the PORT | port number | EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS on success  EMAC\_DRV\_RESULT\_IOCTL\_ERR on failure |
| EMAC\_IOCTL\_HOST\_TX\_RATE\_LIMITER\_CTRL  **For future deliverable, subject to change.** | TBD | TBD | TBD |
| EMAC\_IOCTL\_HOST\_RX\_RATE\_LIMTER\_CTLR  **For future deliverable, subject to change.** | TBD | TBD | TBD |
| EMAC\_IOCTL\_PKT\_TO\_FLOW\_CLASSI\_CTRL  **For future deliverable, subject to change.** | TBD | TBD | TBD |

Table 3 Single Instance ICSS Dual MAC FW IOCTL Commands

#### DUAL MAC Use Case (Interposer Instance ICSS FW)

### Placeholder for future development.

### Platform Specific functions/configuration

Emac\_soc\_v5.c contains AM65XX SOC specific configuration which includes register address mapping, interrupts, NAVSS/UDMAP receive and transmit UDMA channel configuration. The SOC configuration structure will be defined in emac\_soc\_v5.h.

For details of the UDMA subsystem, please refer to Migrating\_Applications\_from\_EDMA\_to\_UDMA\_using\_TI-RTOS .pdf as listed in the reference section.

### Interrupts

Interrupt registration for receive packet is done within the LLD at time of *emac\_open* which uses UDMA event registration API. Once interrupt is received, application provided receive packet callback is invoked.

NOTE: Interrupt support is currently available for DUAL MAC use case, support for interrupts at UDMA ring events is required for SWITCH use case and is being tracked by PRSDK-3812.

### Multi- Core Support

Still an open issue, most likely APIs will be provided to clone driver context (handles) from master core and deliver to secondary cores for use with common APIs to enqueue/dequeue packets. Being tracked by PRSDK-5052 (am65xx: UDMA LLD: How to run instance of LLD on multiple cores, share handles, etc)

### Interposer Card Support

Interposer card is an Ethernet wiring adapter to let 2 icss-g subsystems (instances 0 and 1) drive 2 Ethernet ports with dual mac or switch firmware. This allows the power of more ICSS cores to be applied to each port and direction (RX/TX).

The interposer card divides RGMII RX and TX pins for 2 ports and routes them to separate icss-g RGMII pins as follows:

Interposer eth0 -> RX => icss\_g instance 0, slice 0   (RX only)  => EMAC LLD port 0

interposer eth0 -> TX => icss\_g instance 1, slice1  (TX only) => EMAC LLD port 3

interposer eth1 -> RX => icss\_g instance 1, slice 0   (RX only) => EMAC LLD port 2

interposer eth1 -> TX => icss\_g instance 0, slice 1  (TX only) => EMAC LLD port 1

To support this card with NDK (or 3rd party stacks), EMAC LLD provides two 'virtual' ports:  EMAC7 (virtual port 7) and EMAC8 (virtual port 8) to be used in dual-EMAC mode. For switch mode, three virtual ports are used: 9,10,11. Under the hood, the EMAC\_LLD will treat handling of these virtual ports as follows:

#### Switch Use Use Case (Switch F/W for Interposer Card)

The following virtual ports should be used when invoking API calls for switch use case:

1. *cmac\_open* and *emac close:* Use EMAC\_SWITCH\_PORT (virtual port 12); API will internally open LLD ports 0,1,2 and 3 with reduced configuration.
2. *emac\_poll\_ctrl:*
   1. To poll ETH0 (1st port of switch), use EMAC\_SWITCH\_PORT1; API will internally poll DMA rings associated with (ICSS-G instance 0), LLD port 0 for RX packets and RX MGMT responses. For TX completion events, LLD will internally poll DMA rings associated with (ICSS-G instance 1), LLD port 2.
   2. To poll ETH1 (2nd port of switch), use EMAC\_SWITCH\_PORT2; API will internally poll DMA rings associated with (ICSS-G instance 1) LLD port 2 for RX packets and RX MGMT response. For TX completion events, LLD will internally poll DMA rings associated with (ICSS-G instance 0), LLD port 0.
   3. .
3. *emac\_send*:
   1. to direct packet out of ETH0, use EMAC\_SWITCH\_PORT1; API will internally send directed packet to ICSS instance 1, using DMA resources of LLD port 2.
   2. to direct packet out of ETH1, use EMAC\_SWITCH\_PORT2; API will internally send directed packet out of ICSS instance 0, using the DMA resources of LLD port 0
   3. to send un-directed packet, use EMAC\_SWITCH\_PORT; API will clone the packet and send to both instances using DMA resources of LLD ports 0 and 2. Firmware will figure out if packet needs to be physically transmitted out ETH0/ETH1 or both based on a destination mac address lookup into the Forwarding Database.
4. *emac\_ioctl*: Please refer to the IOCTL table for details about which virtual ports to use.
5. *emac\_get\_statistics\_icssg:* 
   1. to query for EH0, use EMAC\_SWITCH\_PORT1, API will internal query for LLD port 2 for RX and LLD port 3 for TX.
   2. to query for EH1, use EMAC\_SWITCH\_PORT2, API will internal query for LLD port 0 for RX and LLD port 1 for TX.

#### Dual MAC Use Case (Interposer Card with Standalone Dual Mac F/W)

Virtual port 7 handling:

1. *emac\_open* and emac close: internally open (ICSS Instance 0) LLD port 0 and (ICSS instance 1) LLD port 3 with reduced configuration. Since LLD port 0 is for RX handling only, no UDMA TX channels/rings need to be configured at time of *emac\_open*. Similarly for LLD port 3 which is for TX handling only, no UDMA TX channels/rings need to be configured.
2. *emac\_poll\_pkt*: internally poll packets from DMA resources of LLD port 0.
3. *emac\_poll\_ctrl:* internally poll packets from DMA resources of LLD port 0.
4. *emac\_send*: internally send on DMA resources of LLD port 3.
5. *emac\_ioctl*: internally do IOCTL configuration using DMA resources of LLD port 0 as IOCTL is for RX path configuration at this time.
6. *emac\_get\_statistics\_icssg*: internally query for RX statistics from LLD port 0, TX statistics from LLD port 3

Virtual port 8 handling:

1. *emac\_open* and emac close: internally open (ICSS Instance 1) LLD port 2 and (ICSS Instance 0) LLD port 1 with reduced configuration. Since LLD port 2 is for RX handling only, no UDMA TX channels/rings need to be configured at time of *emac\_open*. Similarly for LLD port 1 which is for TX handling only, no UDMA TX channels/rings need to be configured at time of *emac\_open*.
2. *emac\_poll\_pkt*: internally poll packets from DMA resources of LLD port 2.
3. *emac\_poll\_ctrl*: internally poll packets from DMA resources of LLD port 2.
4. *emac\_send*: internally send on port 1.
5. *emac\_ioctl*: internally do IOCTL configuration on LLD port 2 as IOCTL is for RX path configuration at this time
6. *emac\_get\_statistics\_icssg*: internally query for RX statistics from LLD port 2, TX statistics from LLD port 1.

## EMAC Polling Link Status

The application should poll the EMAC periodically (for example every 100msec) to monitor the PHY link status change via the MDIO peripheral using the API *emac\_poll*(). This should make sure any changes in the link status (link up or down) should be communicated to the other modules using the EMAC LLD. The application can disable the polling for link status in the *emac\_open*() API by disabling the MDIO module. Note that this does not apply for Maxwell and MDIO module is always enabled in order to poll for link status.

Function Declaration

*EMAC\_DRV\_ERR\_E emac\_poll(uint32\_t port\_num, EMAC\_LINK\_INFO\_T\* p\_info)*

Where

*typedef struct EMAC\_LinkInfo\_s*

*{*

*bool link\_status\_change;*

*/\*\*< True: link status changed, False: link status is not changed \*/*

*EMAC\_LINK\_STATUS\_T link\_status;*

*/\*\*< PHY link status, only valid when link\_status\_change is TRUE \*/*

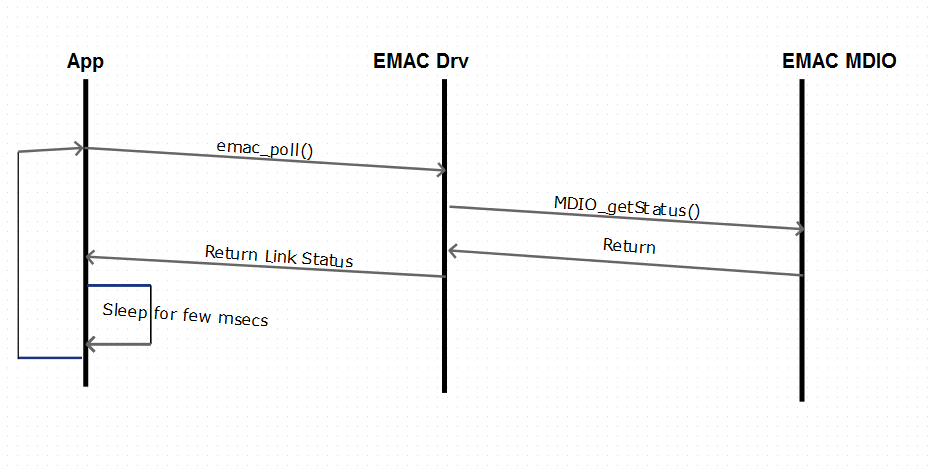
*} EMAC\_LinkInfo;*

For Maxwell EMAC driver use case with ICSSG Switch, when *link\_status\_change* is TRUE, the application will need to convey the link status change to the firmware using EMAC\_IOCTL\_PORT\_STATE\_CTRL as follows:

If *link\_status* is zero (link down), issue EMAC\_IOCTL\_PORT\_STATE\_CTRL with sub-command EMAC\_IOCTL\_PORT\_STATE\_DISABLE.

If *link\_status* is non-zero(link is up), issue EMAC\_IOCTL\_PORT\_STATE\_CTRL with sub-command EMAC\_IOCTL\_PORT\_STATE\_FORWARD or EMAC\_PORT\_BLOCK or EMAC\_PORT\_FORWARD\_WO\_LEARNING as appropriate.

An example for using *emac\_poll* function is shown below



## Error Handling

Error handling is done inside all the LLD APIs and returns following error codes as applicable.

|  |  |
| --- | --- |
| **Error status** | **Description** |
| EMAC\_DRV\_RESULT\_OK | Indicates successful API call. |
| EMAC\_DRV\_RESULT\_GENERAL\_ERR | Generic error status code returned or an unspecified error |
| EMAC\_DRV\_RESULT\_INVALID\_PORT | Invalid EMAC port number error returned from EMAC APIs |
| EMAC\_DRV\_RESULT\_NO\_CHAN\_AVAIL | Error indicating that there is no channels are available. It is returned form EMAC\_init() API |
| EMAC\_DRV\_RESULT\_NO\_MEM\_AVAIL | Error indicating that there is no free memory available. Returned from EMAC\_init APIs |
| EMAC\_DRV\_RESULT\_OPEN\_PORT\_ERR | Error returned from *EMAC\_open* API. |
| EMAC\_DRV\_RESULT\_CLOSE\_PORT\_ERR | Error returned from *EMAC\_close* API |
| EMAC\_DRV\_RESULT\_CONFIG\_PORT\_ERR | Error returned from *EMAC\_config* API |
| EMAC\_DRV\_RESULT\_SEND\_ERR | Error returned from *EMAC\_send* API |
| EMAC\_DRV\_RESULT\_POLL\_ERR | Error returned form *EMAC\_poll* API to indicate poll link status error |
| EMAC\_DRV\_RESULT\_GET\_STATS\_ERR | Error returned from *emac\_get\_statistics* and emac\_get\_statistics\_icssg APIs |
| EMAC\_DRV\_RESULT\_ISR\_ERR | Interrupt service error form emac\_int\_service |
| EMAC\_DRV\_RESULT\_IOCTL\_ERR | IOCTL command error |
| EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS | IOCTL command error, IOCLT command already in progress |
| EMAC\_DRV\_RESULT\_IOCTL\_ERR\_IN\_PROGRESS | VLAN ID is specified in IOCTL is invalid, out of range, valid rang is 0 to 4095 |
| EMAC\_DRV\_RESULT\_IOCTL\_ERR\_PORT\_CLOSED | IOCTL command error, port is closed |
| EMAC\_DRV\_RESULT\_IOCTL\_ERR\_SEND\_MGMT\_MSG | Error when sending MGMT message over PSI I/F to FW |
| EMAC\_DRV\_RESULT\_IOCTL\_IN\_PROGRESS | Successful IOCTL API call and IOCTL command is in progress |

# Standards, Conventions and Procedures

## Documentation Standards

Doxygen format is used for documentation in source code.

## Naming conventions

Processor SDK standard naming conventions are used for file and module naming.

## Programming Standards

* C99 standard data types are used in driver implementation.
* MISRA-C coding standards are followed wherever applicable.

## Software development tools

* TI’s Code Composure Studio for project build setup.
* Make files for source code compilation and Test Applications
* Doxygen for extracting documentation from source code
* Klocworks for static code analysis

# IP Feature List Comparison

This section gives the details of feature comparison of different EMAC HW IPs and software support for those IP features.

NOTE: Table entries below marked with “\*” are supported but currently not tested.

|  |  |
| --- | --- |
| **EMAC IP Features** | |
|  | | **OMAPL137** | | **K2G** | |
|  | | **HW** | **SW** | **HW** | **SW** |
| IP Driver Version | | NA | 0 | NA | 1 |
| No. of hardware instance | | 1 | NA | 1 | NA |
| Synchronous operations. | 10 Mbps | YES | YES | YES | YES \* |
| 100 Mbps | YES | YES | YES | YES \* |
| 1000 Mbps | NO | NA | YES | YES |
| Standard Media Independent Interface (MII) | | YES | YES | YES | YES \* |
| Reduced Media Independent Interface (RMII) | | YES | YES | YES | YES |
| GMII | | NO | NA | NO | NA |
| RGMII | | NO | NA | YES | YES |
| Support quality-of-service (QOS) | | 2 | YES | 8 | YES \* |
| Ether-Stats and 802.3-Stats statistics gathering. | | YES | NA | With RMON  Statistic gathering | NA |
| Transmit CRC generation | | YES | NO | YES | NO |
| Broadcast and Multicast frames selection | | YES | YES | YES | YES \* |
| Promiscuous receive mode | | YES | YES | YES | YES |
| Flow control Support | | YES | YES | YES | YES |
| Programmable interrupt logic | | YES | NA | YES | NA |
| CPPI buffer descriptor memory | | 8k | NA | 2k | NA |
| MDIO module for PHY Management |  | YES | YES | YES | YES |
| Wire rate switching (802.1d) |  | NO | NA | NO | NA |
| Address Lookup Engine (ALE) | address entries plus VLANs | NO | NA | 64 | NA |
| Wire rate lookup | NO | NA | YES | NO |
| Host controlled  Time-based aging | NO | NA | YES | NO |
| Multiple spanning Tree support | NO | NA | YES | NO |
| MAC authentication  (802.1x) | NO | NA | YES | NO |
| MAC address blocking | NO | NA | YES | NO |
| Source port locking | NO | NA | YES | NO |
| OUI host accept/deny  Feature | NO | NA | YES | NO |
| VLAN support | | NO | NA | YES | NO |
| Digital loopback and FIFO loopback modes supported | | NO | NA | YES | NO |
| Emulation Support | | NO | NA | YES | TBD |
| RAM Error Detection and Correction (SECDED) | | NO | NA | YES | NO |
| Programmable transmit Inter-Packet Gap (IPG) | | NO | NA | YES | TBD |

|  |  |
| --- | --- |
| **EMAC IP Features** | |
|  | | **AM335x** | | **AM437x** | | **AM572x** | | **AM6x** | |
|  | | **HW** | **SW** | **HW** | **SW** | **HW** | **SW** | **HW** | **SW** |
| IP Driver Version | | NA | 4 | NA | 4 | NA | 4 | NA | 5 |
| No. of hardware instance | | 2 | NA | 2 | NA | 2 | NA | 1 | NA |
| Synchronous operations. | 10 Mbps | YES | YES | YES | YES | YES | YES | YES | YES \* |
| 100 Mbps | YES | YES | YES | YES | YES | YES | YES | YES \* |
| 1000 Mbps | YES | YES | YES | YES | YES | YES | YES | YES |
| Standard Media Independent Interface (MII) | | NO | NA | NO | YES | NO | YES | NO | NO |
| Reduced Media Independent Interface (RMII) | | YES | YES | YES | YES | YES | YES | YES | NO |
| GMII | | YES | YES | YES | YES | YES | YES | NO | NO |
| RGMII | | YES | YES | YES | YES | YES | YES | YES | YES |
| Support quality-of-service (QOS) | | 4 | YES | 4 | YES | 4 | YES | 8 | YES \* |
| Ether-Stats and 802.3-Stats statistics gathering. | | With RMON  Statistic gathering | NA | With RMON  Statistic gathering | NA | With RMON  Statistic gathering | NA | With RMON  Statistic gathering | NA |
| Transmit CRC generation | | NO | NA | NO | NA | NO | NA | YES | NO |
| Broadcast and Multicast frames selection | | YES | YES | YES | YES | YES | YES | YES | YES |
| Promiscuous receive mode | | YES | YES | YES | YES | YES | YES | YES | YES |
| Flow control Support | | YES | YES | YES | YES | YES | YES | YES | NO |
| Programmable interrupt logic | | YES | NA | YES | NA | YES | NA | YES | NA |
| CPPI buffer descriptor memory | | 8k | NA | 8k | NA | 8k | NA | NA | NA |
| MDIO module for PHY Management |  | YES | YES | YES | YES | YES | YES | YES | YES |
| Wire rate switching (802.1d) |  | YES | NO | YES | NO | YES | NO | NO | NA |
| Address Lookup Engine (ALE) | address entries plus VLANs | 1024 | NA | 1024 | NA | 1024 | NA | 64 | NA |
| Wire rate lookup | YES | NO | YES | NO | YES | NO | YES | NO |
| Host controlled  Time-based aging | YES | NO | YES | NO | YES | NO | YES | NO |
| Multiple spanning Tree support | YES | NO | YES | NO | YES | NO | YES | NO |
| MAC authentication  (802.1x) | YES | NO | YES | NO | YES | NO | YES | NO |
| MAC address blocking | YES | NO | YES | NO | YES | NO | YES | NO |
| Source port locking | YES | NO | YES | NO | YES | NO | YES | NO |
| OUI host accept/deny  Feature | YES | NO | YES | NO | YES | NO | YES | NO |
| VLAN support | | YES | NO | YES | NO | YES | NO | YES | NO |
| Digital loopback and FIFO loopback modes supported | | YES | NO | YES | NO | YES | NO | YES | NO |
| RAM Error Detection and Correction (SECDED) | | NO | NA | NO | NA | NO | NA | NO | NA |
| Programmable transmit Inter-Packet Gap (IPG) | | YES | NO | YES | NO | YES | NO | YES | NO |

# System Design

## Design Approach

The EMAC driver provides a well-defined API layer which allows applications to use the EMAC peripheral to control the flow of packet data from the processor to the PHY and the MDIO module to control PHY configuration and status monitoring.

The EMAC driver is designed to meet the following requirements:

* Support multiple EMAC ports (if available on the device) per core (i.e. A53/R5)
* Support multiple channels per core.
* Support multiple cores to use different channels on the same EMAC port.
* The driver is OS independent and exposes all the operating system callouts via the OSAL layer.
* EMAC example test application provides standard configurations and demonstrates measurable benchmarks.

Platform specific functions are mapped to the platform independent APIs using function table which is given below

/\*! Function to open the specified EMAC port \*/

EMAC\_OpenFxn openFxn;

/\*! Function to config the specified EMAC port for RX filtering, multicast addresses \*/

EMAC\_ConfigFxn configFxn;

/\*! Function to close the specified peripheral \*/

EMAC\_CloseFxn closeFxn;

/\*! Function to send packet to network on specified EMAC port \*/

EMAC\_SendFxn sendFxn;

/\*! Function to poll link status for specified EMAC port\*/

EMAC\_PollFxn pollFxn;

/\*! Function to get EMAC CPSW port statistics\*/

EMAC\_GetStatsFxn getStatsFxn;

/\*! Function to poll for receive packets specified EMAC port\*/

EMAC\_PollPktFxn pollPktFxn;

/\*! Function to get EMAC ICSSG port statistics, IP version 5 only\*/

EMAC\_GetStatsIcssgFxn getStatsIcssgFxn;

/\*! Function to send IOCTL command for specified port, IP version 5 only\*/

EMAC\_IoctlFxn ioctlFxn;

/\*! Function to Poll the driver for specified flow/rings, IP version 5 only \*/

EMAC\_PollCtrlFxn pollCtrl;

## Dependencies

None

## Decomposition of System

The following is an architecture figure which showcases the EMAC driver architecture:-

Platform specific functions/configurations

CSL Register Layer

OSAL Interface

Platform independent APIs

Figure 8 : EMAC LLD Subsystem Block Diagram

The figure illustrates the following key components:-

### Platform Independent APIs

EMAC LLD exposes a set of well-defined APIs which are platform independent and common across the platforms. These are the functions which are exposed to application programs.

### Platform specific functions/configurations

Platform specific functions implement actual functionality of EMAC LLD for a given platform. These functions can be specific to one or set of platforms. There will be multiple versions of platform specific functions based on the number of platforms supported.

Platform specific configurations will define high-level configurations specific to each platform. This includes register address mapping, interrupts, function table initialization etc. These configurations are included in soc file.

### Operating System Abstraction Layer (OSAL)

The EMAC LLD is OS independent and exposes all the operating system callouts via this OSAL layer.

### CSL Functional Layer

The EMAC driver uses the CSL EMAC functional layer to program the device IP by accessing the MMR.

### CSL Register Layer

The register layer is the IP block memory mapped registers which are generated by the IP owner. The EMAC driver does not directly access the MMR registers but uses the EMAC CSL Functional layer for this purpose.

# OMAPL13x Integration

This section describes the changes required for adding OMAPL13x platform support to EMAC LLD.

v0 version of EMAC LLD supported on C6657 platform will be used as reference for the OMAPL13x integration.

## Platform Independent API

There will be no change to platform independent APIs during OMAPL13x integration.

## Platform Specific functions/configuration

EMAC\_soc.c file will be added which defines platform specific configurations for OMAPL13x.

**Gigabit support**

There is gigabit speed support for OMAPL13x platform but existing v0 EMAC driver supports gigabit mode through SGMII. Need to create new version of driver based on v0 for OMAPL13x if we need to avoid SOC specific defines in the driver.

**DNUM dependencies**

DNUM register is used in the EMAC LLD to decide the core number. OMAPL13x platform DNUM register returns a value 1 even though there is only one DSP core which is different from other platforms. LLD changes are needed to handle this case.

## OSAL

No changes are expected for EMAC LLD OSAL for integration of OMAPL13x platform.

## CSL

New version of CSL-RL file is added for OMAPL13x platform.

## Build Setup

Update make files to add support for OMAPL13x platform.