20450 Century Boulevard
Germantown, MD 20874

EMAC Low Level Driver
Software Design Document

Revision 0.7

November 27, 2018

Document License

Contributors to this document

	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

TABLE OF CONTENTS

1. 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.

2. 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.

3. Assumptions

NA

4. 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

Table 1 : Abbreviations and acronyms

5. 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)

6. Design Constraints

6.1 External Constraints / Features

- EMAC LLD should access OS components only through OSAL.

6.2 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.

6.3 Internal Constraints / Requirements

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

7. System Overview

7.1 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.

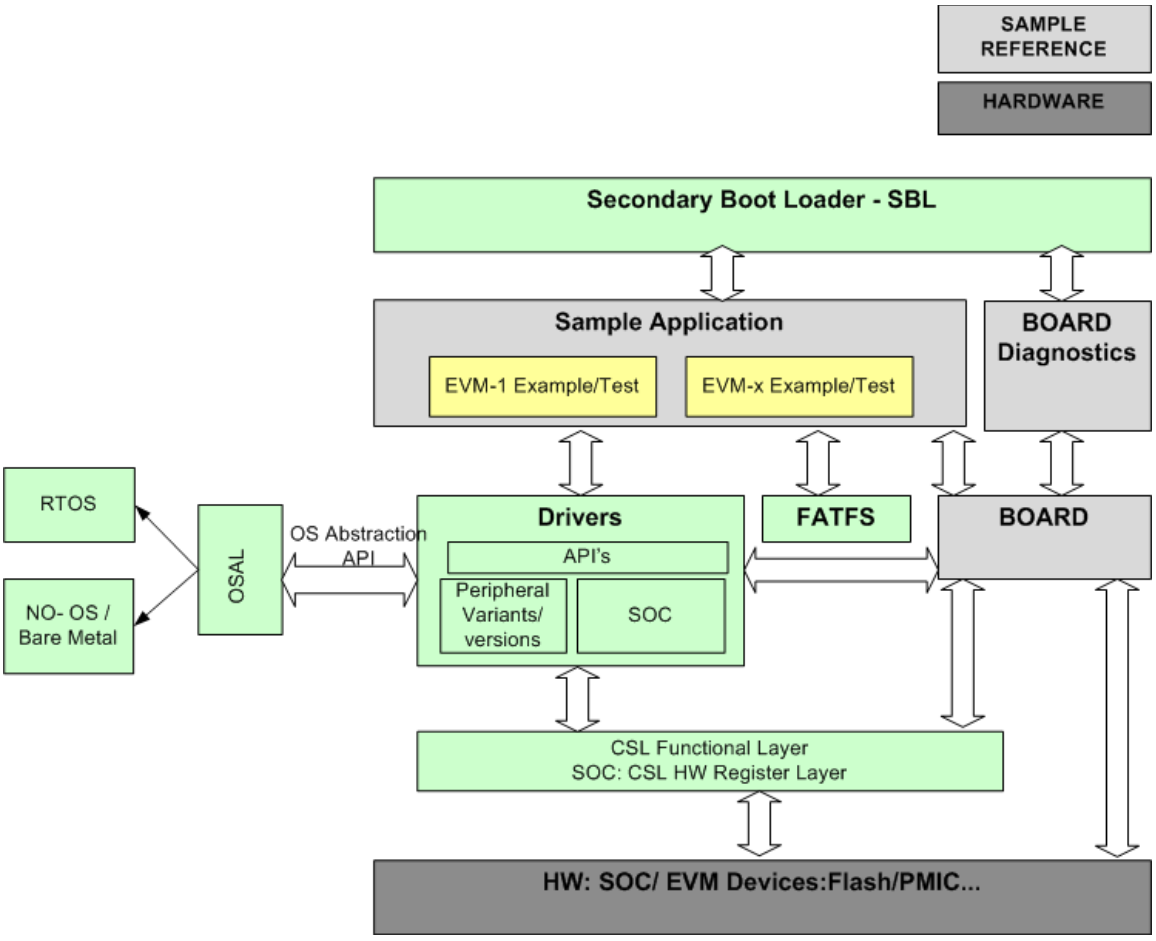


Figure 1 : Process SDK driver subsystem architecture

7.2 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:

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

7.3.1 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:-

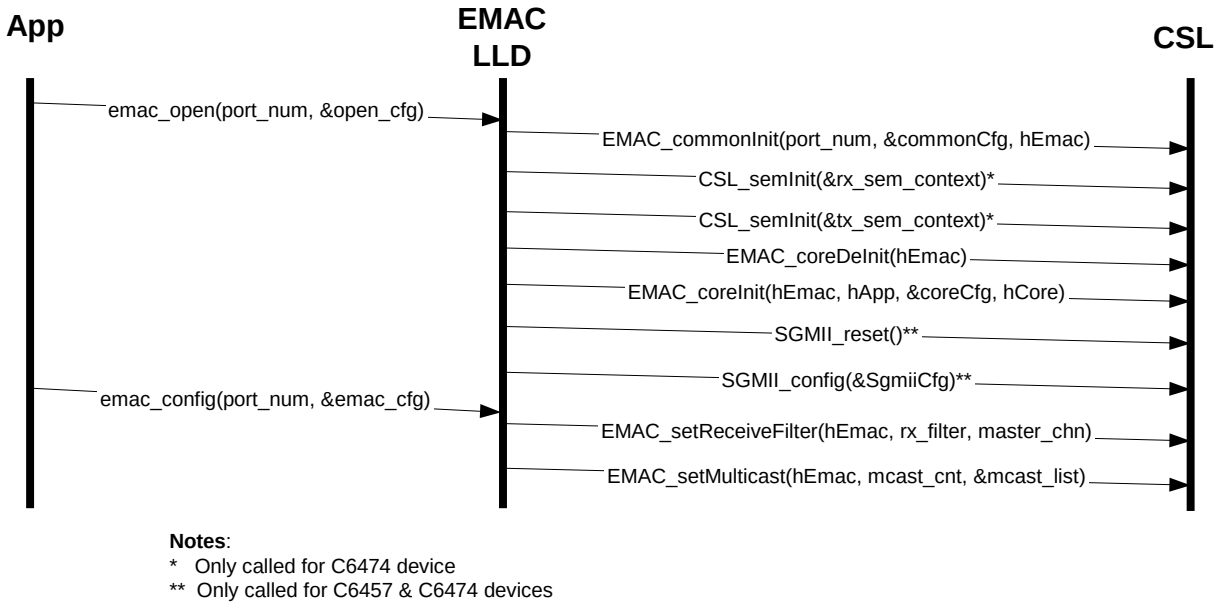


Figure 2 : EMAC configuration

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.

7.3.2 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.

7.3.3 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.

7.3.4 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:-

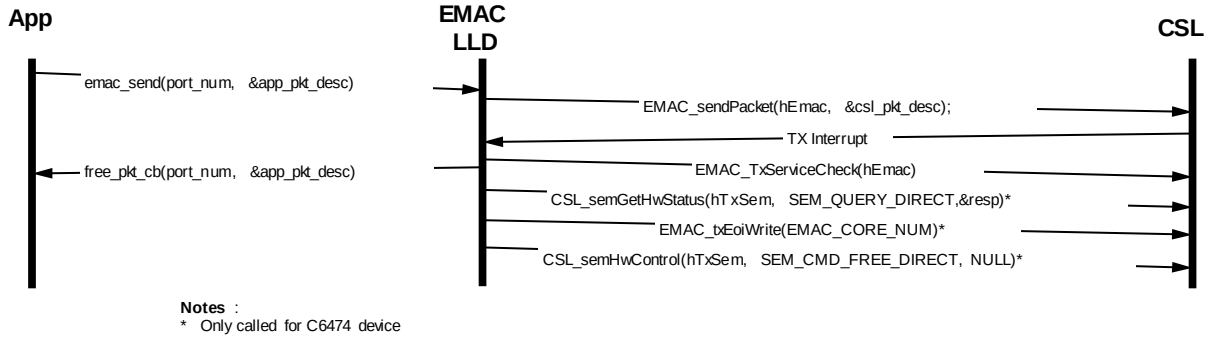


Figure 3 : EMAC TX function

7.3.5 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:-

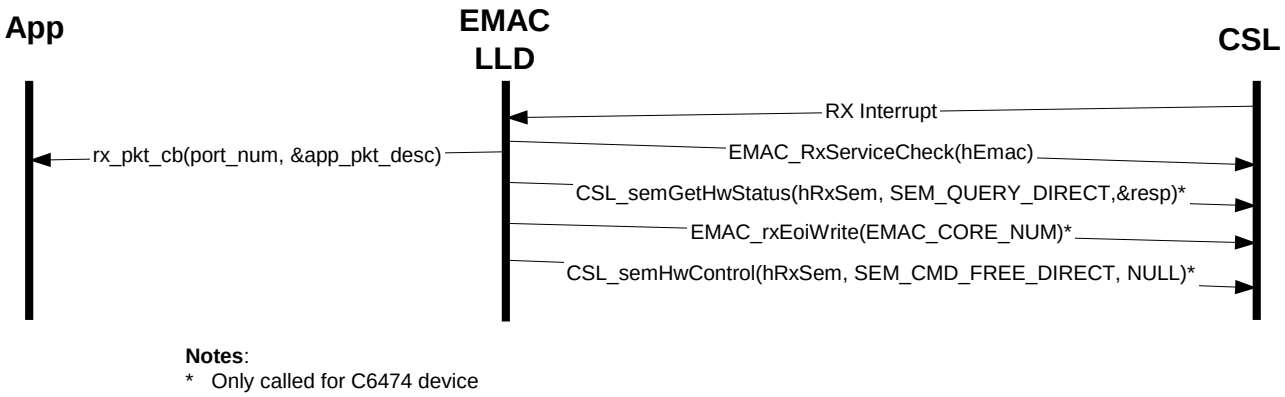


Figure 4 : EMAC RX function

7.3.6 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.

7.3.7 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:-

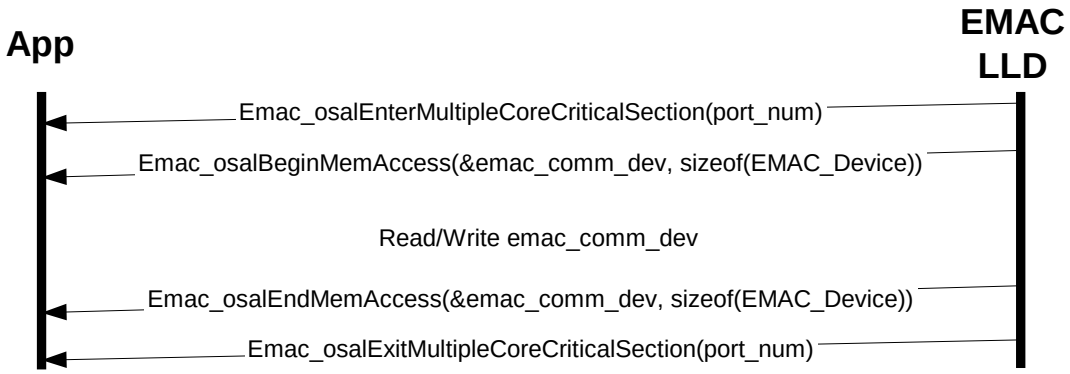


Figure 5 : EMAC Critical section access

7.3.8 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.

7.4 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
    a. RX Packet Channel: EMAC LLD receives packets from the network via ICSSG/CPSW subsystem
    b. RX Config Channel EMAC LLD receives configuration packets (for example configuration 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
- Configuration and related responses from firmware to EMAC LLD for Slice 0 (logical 'half' of an ICSS)
- Physical Port 1 data packets from network
- Configuration 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 via different flows. Note that when a channel is created using the UDMA driver, the default flow is created by "default" and is considered by the driver is the 1st sub-channel or flow.
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 Config Response Channel with M sub-channels per slice.

2. 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. More details about transmitting packets in subsequent sections below. Note the highest priority TX channel (i.e., channel 3) is used to carry configuration messages to firmware as well.

7.4.1 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.

7.4.2 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 test 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.

7.4.3 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 port number 9 to send on port 0 of the switch and use port 10 to send on port 1 of the switch. For un-directed packets use virtual port 11. 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 TxPortQPrior 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. In addition, The application can set the TxPortQPrior field to 0xFF and in this case, the firmware will determine which port queue to use based on the priority REGEN(remap) and TCI 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 code to the calling application in case the port is closed or the driver is unable to submit the packet for transmission on the transmit ring due to the ring being full. 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.

7.4.4 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 3 means are provided by the driver to provide the software descriptor back to the calling application.

1. EMAC_MODE_INTERRUPT: *emac_poll_pkt*() API will PEND on a SEMAPHORE which is posted by TX ISR(ISR registration/SEMAPHORE creation done at time of *emac_open*), invoke TX 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 TX completion queue and invoke the TX callback is packet is present in completion queue. Will return to user application after each *emac_poll_ctrl*() call.
3. Return queue processing at time of next emac_send().Ownership of the passed in descriptor is returned to the application at time of next call to emac_send. The mode of operation can be configured at time of *emac_open()*

7.4.5 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 also need to register a callback at time of emac_open which driver will call to provide the timestamp.

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
3. Register at time of emac_open to receive TX timestamp response by using tx_ts_cb field of EMAC_OpenConfigInfo. Prototype of tx_ts-cb: uint32_t port_num, uint32_t ts_id, uint64_t time_stamp
4. In order to retrieve the timestamp, the application will need to use the *emac_poll_ctrl* as follows:
    a. emac_poll_ctrl(port_num, rxPktRings, rxCfgRings, txRings) where rxCfgRings is a bitmap of RX Config rings to poll needs to have bit 2 set. If timestamp Config packet is available, tx_ts_cb will be invoked.

7.4.6 Packet RX

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. 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.

7.4.7 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.

7.4.8 New APIs

API to retrieve ICSS-G hardware statistics:

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 configuration response ring and tx completion rings.

2. EMAC_DRV_ERR_E emac_poll_ctrl(uint32_t port_num, uint32_t rxPktRings, uint32_t rxCfgRings, uint32_t txRings)

   rxPktRings is a bitmap of which packet completion rings to poll.
   rxCfgRings is a bitmap of which configuration rings to poll.
   txRings is a bitmap of which packet transmit completion rings to poll.

API for issuing IOCTL commands for ICSSG ports

3. EMAC_DRV_ERR_E *emac_ioctl*(uint32_t port_num, uint32_t emacIoctlCmd, void *emacIoctlParams)

The table below provides a list of IOCTLS supported for ICSSG use case:

IOCTL Command/Sub Command	Description	IOCTL Parameters	RETURN TYPE
EMAC_IOCTL_PROMISCOUS_MODE_CTRL (MMR update, non-blocking)	Enable/disable promiscuous mode of operation (dual-mac use case, switch TBD)	Port number, ENABLE(1), DISABLE(0)	EMAC_DRV_RESULT_OK on success EMAC_DRV_RESULT_GENERAL_ERR on failure (invalid port)
EMAC_IOCTL_UNICAST_FLOOD_CTRL (CFG message to FW over PSI I/F, blocking call)	Enable/disable flooding of unknown unicast packets to host port	Port number, ENABLE(1), DISABLE(0)	EMAC_DRV_RESULT_OK on success EMAC_DRV_RESULT_GENERAL_ERR on failure (invalid port)
EMAC_IOCTL_FDB_ENTRY_CTRL/ EMAC_IOCTL_FDB_ENTRY_ADD (CFG message to FW over PSI I/F, blocking call)	Add forward data base entry to internal ICSSG memory.	Switch number, 6 byte mac address, vlan_tag, fdb_entry	EMAC_DRV_RESULT_OK on success, EMAC_DRV_RESULT_GENERAL_ERR
EMAC_IOCTL_FDB_ENTRY_CTRL/ EMAC_IOCTL_FDB_ENTRY_DEL (CFG message to FW over PSI I/F, blocking call)	Delete forward data base entry from internal ICSSG memory	Switch, number, 6 byte mac address, vlan_tag, fdb_entry	EMAC_DRV_RESULT_OK on success, EMAC_DRV_RESULT_GENERAL_ERR (due to timeout)
EMAC_IOCTL_FDB_ENTRY_CTRL/ EMAC_IOCTL_FDB_ENTRY_DEBUG (CFG message to FW over PSI I/F, blocking call)	Debug IOCTL for retrieving forward data base entry from internal ICSSG memory.	Switch number, Slot number	EMAC_DRV_RESULT_OK on success, EMAC_DRV_RESULT_GENERAL_ERR (due to timeout) return 32 bytes of data (6 byte mac_addr, 1 byte vlan fid, 1 byte

			fdb_entry, 4 entries per slot) on success
EMAC_IOCTL_PORT_CTRL (CFG message to FW over PSI I/F, blocking call)	Used to set various port states.	Switch number, Port_number, Port_state (disabled, blocking, forward, forward without learning)	EMAC_DRV_RESULT_OK on success, EMAC_DRV_RESULT_GENERAL_ERR (due to timeout)
EMAC_IOCTL_SAV_CHECK_CTRL (CFG message to FW over PSI I/F, blocking call)	Source address violation check enable/disable control.	Switch number, Port_number, ENABLE(1), DISABLE(0)	EMAC_DRV_RESULT_OK on success, EMAC_DRV_RESULT_GENERAL_ERR (due to timeout)
EMAC_IOCTL_VLAN_CTRL/ EMAC_IOCTL_VLAN_SET_DEFAULT_TBL (MMR update, non-blocking)	Update ICSSG shared memory with default vlan table entries (4096 entries set to default settings)	Switch number, vlan_fid, vlan_info	EMAC_DRV_RESULT_OK on success EMAC_DRV_RESULT_GENERAL_ERR on failure
EMAC_IOCTL_VLAN_CTRL/ EMAC_IOCTL_VLAN_ADD_ENTRY (MMR update, non-blocking)	Add entry to vlan table for specified vlan tag value where tag is from 0 to 4095).	Switch number, vlan_tag(table entry), vlan_fid, vlan_info	EMAC_DRV_RESULT_OK on success EMAC_DRV_RESULT_GENERAL_ERR on failure
EMAC_IOCTL_VLAN_CTRL/ EMAC_IOCTL_VLAN_AWARE_MODE (MMR update and CFG message to FW over PSI I/F, blocking call)	Enable/disable VLAN aware mode.	Switch number, ENABLE(1), DISABLE(0)	EMAC_DRV_RESULT_OK on success, EMAC_DRV_RESULT_GENERAL_ERR (due to timeout)
EMAC_IOCTL_VLAN_CTRL/ EMAC_IOCTL_VLAN_SET-DEFAULT_TAG (MMR update and CFG message to FW over PSI I/F, blocking call)	Set default tag for switch to use when un-tagged or priority tagged packet is received.	Switch number, Port number, 32-bit tag/priority	EMAC_DRV_RESULT_OK on success, EMAC_DRV_RESULT_GENERAL_ERR (due to timeout)
EMAC_IOCTL_PRIO_REGEN_CTRL (MMR update and CFG message to FW over PSI I/F, blocking call)	Configure the priority regeneration table for a port including the host port.	Switch number, Port number (-1 for host port) ENABLE(1), DISABLE(0), 8-byte REMAP array	EMAC_DRV_RESULT_OK on success, EMAC_DRV_RESULT_GENERAL_ERR (due to timeout)
EMAC_IOCTL_TCI_MAPPING_CTRL ((MMR update, non-blocking)	Configure the mapping of traffic class to port queue.	Switch number, Port number (-1 for host port), 8-byte TCI to portQ mapping	EMAC_DRV_RESULT_OK on success, EMAC_DRV_RESULT_GENERAL_ERR (due to timeout)
EMAC_IOCTL_ACCEPTABLE_FRAM_CHECK_CTRL (MMR update and CFG message to FW over PSI I/F, blocking call)	Configure acceptable frame checking rules per physical port	Switch number, Port number, TBD	EMAC_DRV_RESULT_OK on success, EMAC_DRV_RESULT_GENERAL_ERR (due to timeout)
EMAC_IOCTL_STREAM_FDB_CTRL	TBD	TBD	TBD
EMAC_IOCTL_CUT_THROUGH_CTRL	TBD	TBD	TBD
EMAC_IOCTL_HOST_TX_RATE_LIMITER_CTRL	TBD	TBD	TBD
EMAC_IOCTL_HOST_RX_RATE_LIMTER_CTLR	TBD	TBD	TBD
EMAC_IOCTL_PKT_TO_FLOW_CLASSI_CTRL	TBD	TBD	TBD

7.4.9 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.

7.4.10 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.

Interrupt registration for transmit complete is done within the LLD at time of *emac_open* which uses UDMA event registration API. Once interrupt is received, application provided transmit complete callback is invoked.

7.4.11 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)

7.4.12 Interposer Card Support

Interposer card is an Ethernet wiring adapter to let 2 icss-g subsystems drive 2 Ethernet ports with dual mac or switch firmware. So that each direction (TX/RX) can be handled by one pair of PRU cores. The interposer card permits the power of two icss-g instances to be used on a two port switch or dual EMAC as opposed to the two-port, single icss-g configuration.

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. Under the hood, the EMAC_LLD will treat handling of these virtual ports as follows:

Virtual port 7 handling:

1. *emac_open* and emac close: internally open ports 0 and 3 with reduced configuration. Since port 0 is for RX handling only, no UDMA TX channels/rings need to be configured at time of *emac_open*. Similarly for port 3 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 for port 0.
3. *emac_send*: internally send on port 3.
4. *emac_ioctl*: internally do IOCTL configuration on port 0 as IOCTL is for RX path configuration at this time.
5. *emac_get_statistics_icssg*: internally query for RX statistics from port 0, TX statistics from port 3

Virtual port 8 handling:

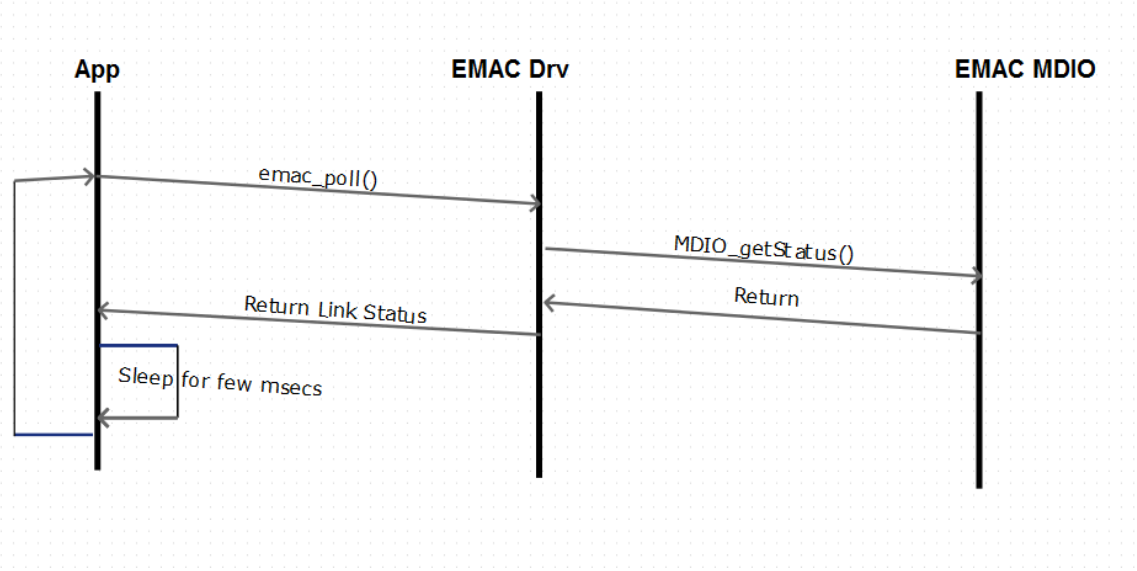
6. *emac_open* and emac close: internally open ports 2 and 1 with reduced configuration. Since port 2 is for RX handling only, no UDMA TX channels/rings need to be configured at time of *emac_open*. Similarly for port 1 which is for TX handling only, no UDMA TX channels/rings need to be configured at time of *emac_open*.
7. *emac_poll_pkt*: internally poll packets for port 2.
8. *emac_send*: internally send on port 1.
9. *emac_ioctl*: internally do IOCTL configuration on port 2 as IOCTL is for RX path configuration at this time.

emac_get_statistics_icssg: internally query for RX statistics from port 2, TX statistics from port 1.

7.5 EMAC Polling Link Status

The application should poll the EMAC periodically (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.

An example for using *emac_poll* function is shown below



7.6 Error Handling

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

Error status	Description
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.

8. Standards, Conventions and Procedures

8.1 Documentation Standards

Doxygen format is used for documentation in source code.

8.2 Naming conventions

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

8.3 Programming Standards

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

8.4 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

9. 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

10. System Design

10.1 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;
```

10.2 Dependencies

None

10.3 Decomposition of System

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

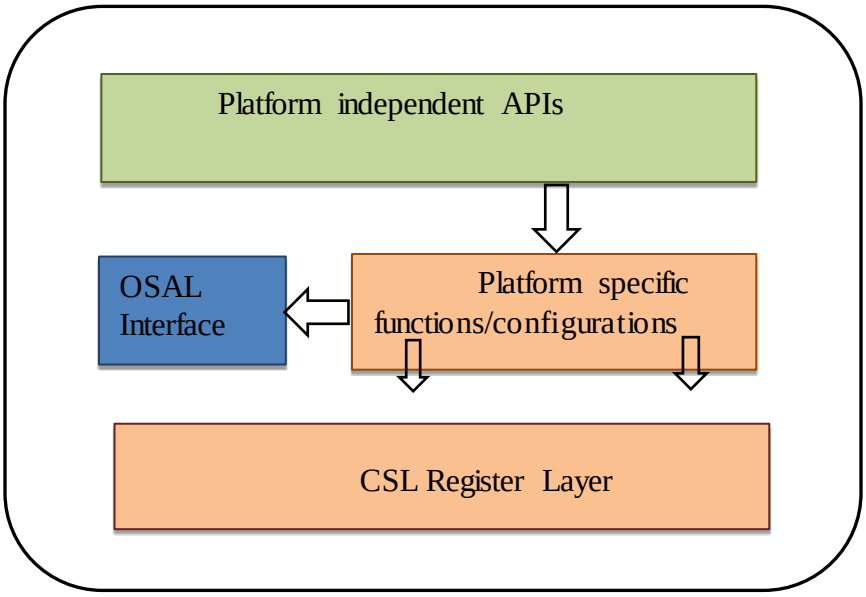


Figure 6 : EMAC LLD Subsystem Block Diagram

The figure illustrates the following key components:-

10.3.1 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.

10.3.2 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.

10.3.3 Operating System Abstraction Layer (OSAL)

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

10.3.4 CSL Functional Layer

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

10.3.5 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.

11. 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.

11.1 Platform Independent API

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

11.2 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.

11.3 OSAL

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

11.4 CSL

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

11.5 Build Setup

Update make files to add support for OMAPL13x platform.