



ICSS DUAL EMAC FIRMWARE DESIGN GUIDE

ICSS based Dual Ethernet MAC

Applies to Product Release: 01.00.00.08
Publication Date: September 6, 2017

Document License

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

Contributors to this document

Copyright (C) 2011-2017 Texas Instruments Incorporated - <http://www.ti.com/>



Texas Instruments, Incorporated
20450 Century Boulevard
Germantown, MD 20874 USA

This document is intended for users who are interested in getting more detailed understanding of the firmware design. It discusses ICSS based EMAC firmware implementation details along with any features added on top of the basic EMAC firmware. It mentions the memory maps, structures and software design flow of the firmware.

Note: Those who just want to use ICSS EMAC firmware may not need to go through this document.

Contents

ICSS based Dual Ethernet MAC	1
Contents.....	3
List of Figures	4
List of Tables.....	5
1 Requirements.....	6
1.1 EMAC High Level Requirements.....	6
1.2 Implementation Constraints	6
2 Design Description	7
2.1 System Decomposition Diagram.....	7
3 Firmware Detailed Design	8
3.1 Firmware Architecture Overview.....	8
3.1.1 <i>Architecture and Design</i>	<i>8</i>
3.1.2 <i>PRU0/1 DMEM and ICSS Shared RAM Memory Map for EMAC Firmware</i>	<i>9</i>
3.1.3 <i>Scratchpad Usage Design</i>	<i>14</i>
3.1.4 <i>PRU Register Usage Design</i>	<i>16</i>
3.1.5 <i>Micro Scheduler Task Design</i>	<i>17</i>
3.1.6 <i>Receive Task Design</i>	<i>19</i>
3.1.7 <i>Buffer Descriptors, Queue Descriptors and Receive Context</i>	<i>24</i>
3.1.8 <i>Quality of Service (QoS).....</i>	<i>26</i>
3.1.9 <i>Transmit Task design</i>	<i>27</i>
3.1.10 <i>Statistics Task</i>	<i>30</i>
3.1.11 <i>Storm Prevention</i>	<i>31</i>
3.1.12 <i>Half Duplex Support</i>	<i>31</i>
3.1.13 <i>Link Status Change detection.....</i>	<i>31</i>
3.1.14 <i>EMAC Time Triggered Send</i>	<i>32</i>
3.2 Firmware Macros Description	43
3.3 Firmware Sources Description	45
4 Revision History	46

List of Figures

Figure 1: Software Architecture	7
Figure 2: System Decomposition Diagram	7
Figure 3: EMAC Firmware High Level Architecture	8
Figure 4: Micro Scheduler Flow Chart	18
Figure 5: Round-robin approach followed by MS	18
Figure 6: RCV_FB Flow Chart	21
Figure 7: RCV_NB Flow Chart	22
Figure 8: RCV_LB Flow Chart	23
Figure 9: XMT_FB Flow Chart	28
Figure 10: XMT_NB Flow Chart	29
Figure 11: XMT_LB Flow Chart	29
Figure 12: Time Triggered Send Overview	32
Figure 13: TTS Flow Chart (Part 1)	38
Figure 14: TTS Flow Chart (Part 2)	39
Figure 15: Incorrect Cyclic Packet Transmission in TTS	41
Figure 16: Time Availability Check in TTS	42

List of Tables

Table 1: High Level Requirements.....	6
Table 3: PRU0 and PRU1 DMEM Memory Map.....	9
Table 4: ICSS Shared RAM Memory Map.....	10
Table 5: L3 OCMC RAM Memory Map.....	11
Table 6: Statistics Offsets	13
Table 7: Scratchpad Register Usage	14
Table 8: PRU Register Usage.....	16
Table 9: Buffer Descriptor Bits	24
Table 10: Queue Descriptor Bits.....	24
Table 11: TTS Source Code Files List.....	32
Table 12: TTS R22 Bits Usage	33
Table 13: TTS Control Variables.....	34
Table 14: TTS Memory Map	34
Table 15: TTS Status Bits	35
Table 16: TTS Functions.....	36
Table 17: TTS Compare Register Usage.....	40
Table 18: ICSS EMAC Macros	44
Table 19: Firmware Sources Description.....	45
Table 20: Revision History	46

1 Requirements

1.1 EMAC High Level Requirements

Requirements	Remarks
1 ms buffering per port	Supported
Host IRQ	Supported
Ethernet QoS	Supported With 2 queues instead of 8. So, it is not a standard Ethernet QoS implementation.
Statistics	Supported
Storm Prevention	Supported

Table 1: High Level Requirements

1.2 Implementation Constraints

Implementation constraints are as follows:

1. Hardware timer resolution is 5 nsec @ IEP clock of 200 MHz, Timer register are updated in jumps of 5 @ each tick (except during clock synchronization) to emulate a 1nSec granularity.
2. Only frames with 8 bytes of preamble are supported.

2 Design Description

This section discusses the overall flow & interaction of EMAC Firmware.

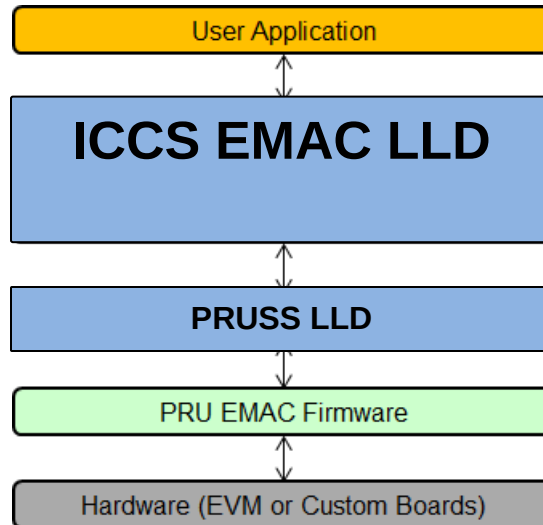


Figure 1: Software Architecture

2.1 System Decomposition Diagram

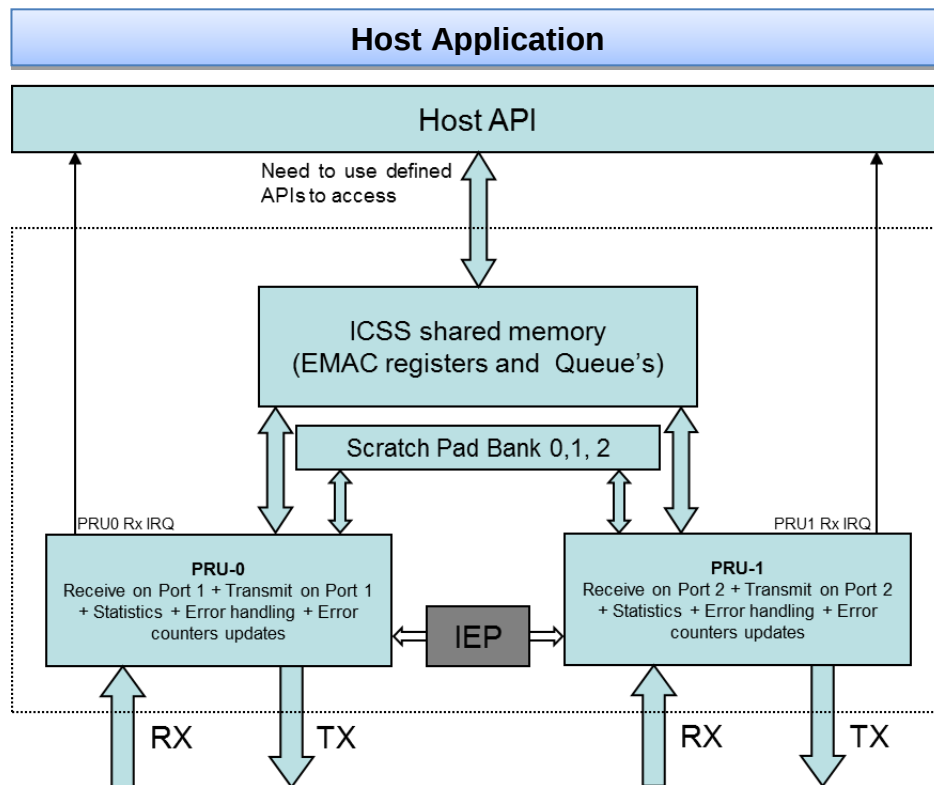


Figure 2: System Decomposition Diagram

3 Firmware Detailed Design

3.1 Firmware Architecture Overview

3.1.1 Architecture and Design

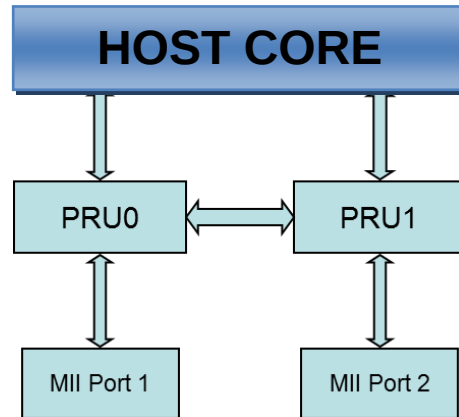


Figure 3: EMAC Firmware High Level Architecture

1. Host Core functions (ARM, DSP)
 - i. Run Host Application
 - ii. EMAC Initializations
2. PRU0 functions
 - i. Receive and transmit frames on Port 1
 - ii. Statistics for received and transmitted frames on Port1
3. PRU1 functions are exactly symmetrical to PRU0.

3.1.2 PRU0/1 DMEM and ICSS Shared RAM Memory Map for EMAC Firmware

3.1.2.1 PRU0 and PRU1 DMEM Memory Map

Memory map usage for DRAM in EMAC is identical for both PRUs and is as shown below.

Definition	Offset	Remarks
Reserved	0x0000 to 0x03FF	Reserved for future use.
Protocol Specific	0x0400 to 0x1E97	Available for protocol specific usage.
ICSS_EMAC_TTS_CYCLE_START_OFFSET	0x1E98	Time Triggered Send related offsets. For details refer EMAC Time Triggered Send section.
ICSS_EMAC_TTS_CYCLE_PERIOD_OFFSET	0x1EA0	
ICSS_EMAC_TTS_CFG_TIME_OFFSET	0x1EA4	
ICSS_EMAC_TTS_STATUS_OFFSET	0x1EA8	
ICSS_EMAC_TTS_MISSED_CYCLE_CNT_OFFSET	0x1EAC	
ICSS_EMAC_TTS_PREV_TX_SOF	0x1EB0	
ICSS_EMAC_TTS_CYC_TX_SOF	0x1EB8	
PORT_QUEUE_DESC_OFFSET	0x1EC0	Port queue descriptors for 4 port queues.
Q1_TX_CONTEXT_OFFSET	0x1EE0	Transmit context for the priority 0 transmit queue on Port 1/2.
Q2_TX_CONTEXT_OFFSET	0x1EE8	Transmit context for the priority 1 transmit queue on Port 1/2.
Q3_TX_CONTEXT_OFFSET	0x1EF0	Transmit context for the priority 2 transmit queue on Port 1/2.
Q4_TX_CONTEXT_OFFSET	0x1EF8	Transmit context for the priority 3 transmit queue on Port 1/2.
Statistics	0x1F00 to 0x1FA8	EMAC Statistics for PRU are stored in this memory space.
Free space	0x1FA9 to 0x2000	This memory space is free.

Table 2: PRU0 and PRU1 DMEM Memory Map

3.1.2.2 ICSS Shared RAM Memory Map

Below are offset addresses of the buffer descriptors for the default configuration of queue sizes.

Definition	Offset	Remarks
Reserved	0x0000 to 0x03FF	Reserved for future use.
P0_Q1_BD_OFFSET	0x0400	Buffer descriptors for the priority 0 host receive queue.
P0_Q2_BD_OFFSET	0x0708	Buffer descriptors for the priority 1 host receive queue.
P0_Q3_BD_OFFSET	0x0A10	Buffer descriptors for the priority 2 host receive queue.
P0_Q4_BD_OFFSET	0x0D18	Buffer descriptors for the priority 3 host receive queue.
P1_Q1_BD_OFFSET	0x1020	Buffer descriptors for the priority 0 transmit queue on Port 1.
P1_Q2_BD_OFFSET	0x11A4	Buffer descriptors for the priority 1 transmit queue on Port 1.
P1_Q3_BD_OFFSET	0x1328	Buffer descriptors for the priority 2 transmit queue on Port 1.
P1_Q4_BD_OFFSET	0x14AC	Buffer descriptors for the priority 3 transmit queue on Port 1.
P2_Q1_BD_OFFSET	0x1630	Buffer descriptors for the priority 0 transmit queue on Port 2.
P2_Q2_BD_OFFSET	0x17B4	Buffer descriptors for the priority 1 transmit queue on Port 2.
P2_Q3_BD_OFFSET	0x1938	Buffer descriptors for the priority 2 transmit queue on Port 2.
P2_Q4_BD_OFFSET	0x1ABC	Buffer descriptors for the priority 3 transmit queue on Port 2.
HOST_Q1_RX_CONTEXT_OFFSET	0x1C40	Receive context for the priority 0 host receive queue.
HOST_Q2_RX_CONTEXT_OFFSET	0x1C48	Receive context for the priority 1 host receive queue.
HOST_Q3_RX_CONTEXT_OFFSET	0x1C50	Receive context for the priority 2 host receive queue.
HOST_Q4_RX_CONTEXT_OFFSET	0x1C58	Receive context for the priority 3 host receive queue.
HOST_QUEUE_DESCRIPTOR_OFFSET_ADDR	0x1C60	Table for host buffer descriptor base offsets for 4 host queues.
HOST_QUEUE_OFFSET_ADDR	0x1C68	Table for host buffer base offsets for 4 host queues.
HOST_QUEUE_SIZE_ADDR	0x1C70	Table for queue sizes of host, port 1 and port 2.
HOST_QUEUE_DESC_OFFSET	0x1C80	Host queue descriptors for 4 host queues.
Protocol Specific	0x1CA0 to 0x1F9F	Available for protocol specific usage.
Free space	0x1FA0 to 0x3000	This memory space is free.

Table 3: ICSS Shared RAM Memory Map

3.1.2.3 L3 OCMC RAM Memory Map

Below are offset addresses of the queue data buffers for the default configuration of queue sizes. Since the queue sizes are configurable, offset addresses will automatically change when the size of one or more queue is changed.

Definition	Offset	Remarks
P0_Q1_BUFFER_OFFSET	0x0000	Offset of data buffers of first priority host queue in L3 RAM
P0_Q2_BUFFER_OFFSET	0x1840	Offset of data buffers of second priority host queue in L3 RAM
P0_Q3_BUFFER_OFFSET	0x3080	Offset of data buffers of third priority host queue in L3 RAM
P0_Q4_BUFFER_OFFSET	0x48C0	Offset of data buffers of fourth priority host queue in L3 RAM
P1_Q1_BUFFER_OFFSET	0x6100	Offset of data buffers of first priority Port1 queue in L3 RAM
P1_Q2_BUFFER_OFFSET	0x6D20	Offset of data buffers of second priority Port1 queue in L3 RAM
P1_Q3_BUFFER_OFFSET	0x7940	Offset of data buffers of third priority Port1 queue in L3 RAM
P1_Q4_BUFFER_OFFSET	0x8560	Offset of data buffers of fourth priority Port1 queue in L3 RAM
P2_Q1_BUFFER_OFFSET	0x9180	Offset of data buffers of first priority Port2 queue in L3 RAM
P2_Q2_BUFFER_OFFSET	0x9DA0	Offset of data buffers of second priority Port2 queue in L3 RAM
P2_Q3_BUFFER_OFFSET	0xA9C0	Offset of data buffers of third priority Port2 queue in L3 RAM
P2_Q4_BUFFER_OFFSET	0xB5E0	Offset of data buffers of fourth priority Port2 queue in L3 RAM
Reserved Space	0xC200 to 0xEDFF	Since queue sizes are programmable, this space acts as buffer if there is need to increase the queue sizes.

Table 4: L3 OCMC RAM Memory Map

3.1.2.4 Statistics memory map

This is common to both PRU0 and PRU1, DRAM0 stores statistics for PRU0 and DRAM1 for PRU1.

Definition	Offset	Remarks
TX_BC_FRAMES_OFFSET	0x1F00	Number of Transmitted Broadcast Frames
TX_MC_FRAMES_OFFSET	0x1F04	Number of Transmitted Multicast Frames
TX_UC_FRAMES_OFFSET	0x1F08	Number of Transmitted Unicast Frames
TX_BYTE_CNT_OFFSET	0x1F0C	Total Number of Bytes transmitted
RX_BC_FRAMES_OFFSET	0x1F10	Number of Received Broadcast Frames
RX_MC_FRAMES_OFFSET	0x1F14	Number of Received Multicast Frames
RX_UC_FRAMES_OFFSET	0x1F18	Number of Received Unicast Frames
RX_BYTE_CNT_OFFSET	0x1F1C	Total Number of Bytes received
TX_64_BYTE_FRAME_OFFSET	0x1F20	Number of transmitted packets of size 64 bytes
TX_65_127_BYTE_FRAME_OFFSET	0x1F24	Number of transmitted packets of size between 65-127 bytes.
TX_128_255_BYTE_FRAME_OFFSET	0x1F28	Number of transmitted packets of size between 128-255 bytes.
TX_256_511_BYTE_FRAME_OFFSET	0x1F2C	Number of transmitted packets of size between 256-511 bytes.
TX_512_1023_BYTE_FRAME_OFFSET	0x1F30	Number of transmitted packets of size between 512-1023 bytes.
TX_1024_MAX_BYTE_FRAME_OFFSET	0x1F34	Number of transmitted packets of size greater than 1023 bytes.
RX_64_BYTE_FRAME_OFFSET	0x1F38	Number of received packets of size 64 bytes
RX_65_127_BYTE_FRAME_OFFSET	0x1F3C	Number of received packets of size between 65-127 bytes.
RX_128_255_BYTE_FRAME_OFFSET	0x1F40	Number of received packets of size between 128-255 bytes.
RX_256_511_BYTE_FRAME_OFFSET	0x1F44	Number of received packets of size between 256-511 bytes.
RX_512_1023_BYTE_FRAME_OFFSET	0x1F48	Number of received packets of size between 512-1023 bytes.
RX_1024_MAX_BYTE_FRAME_OFFSET	0x1F4C	Number of received packets of size greater than 1023 bytes.
LATE_COLLISION_OFFSET	0x1F50	Number of packets which suffered late collision
SINGLE_COLLISION_OFFSET	0x1F54	Number of bytes which suffered only one collision
MULTIPLE_COLLISION_OFFSET	0x1F58	Number of bytes which suffered more than one collision
EXCESS_COLLISION_OFFSET	0x1F5C	Number of bytes which suffered more than 15 collisions
RX_MISALIGNMENT_COUNT_OFFSET	0x1F60	Packets which had an odd number of nibbles
STORM_PREVENTION_COUNTER	0x1F64	Multicast and Broadcast Packets which were discarded by Storm prevention
RX_ERROR_OFFSET	0x1F68	Number of packets which triggered Rx MAC errors or number of instances where a MAC error was detected.
SFD_ERROR_OFFSET	0x1F6C	Number of Packets with incorrect preamble.
TX_DEFERRED_OFFSET	0x1F70	Packets which were deferred from transmission at least once.

TX_ERROR_OFFSET	0x1F74	Reserved
RX_OVERSIZED_FRAME_OFFSET	0x1F78	Number of packets with byte size greater than 1522. (Default value. This is a programmable value in MII RT)
RX_UNDERSIZED_FRAME_OFFSET	0x1F7C	Number of packets with byte size less than 64 (including CRC). Packets with size less than 18 are counted as Rx Error.
RX_CRC_COUNT_OFFSET	0x1F80	Number of Packets with CRC/FCS Error.
RX_DROPPED_FRAMES_OFFSET	0x1F84	Number of frames dropped due to link loss/same dst as host.
TX_OVERFLOW_COUNTER	0x1F88	Number of times TX FIFO overflow occurred.
TX_UNDERFLOW_COUNTER	0x1F8C	Number of times TX FIFO underflow occurred.

Table 5: Statistics Offsets

3.1.3 Scratchpad Usage Design

Three shared scratchpads (10, 11 and 12) with 30 registers each between PRU0 and PRU1 are used for keeping Receive Task and Transmit Task contexts.

Structure for MII TX context scratchpad entry:

```
.struct MII_TX_DESC
    .u8          flags
    .u16         QUEUE_DESC_OFFSET
    .u16         BYTE_CNT
    .u16         Packet_Length
    .u16         BUFFER_DESC_OFFSET
    .u16         BUFFER_INDEX
    .u16         BUFFER_OFFSET
    .u16         TOP_MOST_BUFFER_INDEX
    .u16         BASE_BUFFER_DESC_OFFSET
    .u16         TOP_MOST_BUFFER_DESC_OFFSET
.ends
```

Structure for MII RX context scratchpad entry:

```
.struct MII_RCV_DESC
    .u8          rx_flags
    .u8          tx_flags
    .u8          rx_flags_extended
    .u8          qos_queue
    .u16         byte_cntr
    .u16         wrkng_wr_ptr
    .u16         rd_ptr
    .u16         buffer_index
    .u16         base_buffer_index
    .u16         rcv_queue_pointer
    .u16         base_buffer_desc_offset
    .u16         top_most_buffer_desc_offset
.ends
```

Below table shows the allocation of above TX and RCV contexts on the scratchpad:

PRU Core	BANK0	BANK1	BANK2
PRU0 TX Context		REG 13 to REG 17	
PRU1 TX Context			REG 13 to REG 17
PRU0 Host RCV Context		REG 25 to REG 29	
PRU1 Host RCV Context			REG 25 to REG 29

Table 6: Scratchpad Register Usage

Below are defines used for storing and reading context/data from/to the scratchpad and RX L2 FIFO. These are the Bank IDs.

```
#define BANK0          10
#define BANK1          11
#define BANK2          12
#define RX_L2_BANK0_ID 20
#define RX_L2_BANK1_ID 21
```

3.1.4 PRU Register Usage Design

Each PRU has 32 registers, from REG 0 to REG 31, out of which REG 30 and REG 31 have special use and cannot be used by firmware for storing data. Below is the register allocation table which shows the registers used by various tasks. Since the firmware is symmetrical on both PRUs, same allocation is true for both PRUs.

R22 is used as a persistent register on both PRU's. Bits 15-21 are used by Time Triggered Send implementation while bits 22-31 are used by basic EMAC. Bits 0 to 14 are free for usage by other protocols. For more details, refer to function and macro descriptions in firmware source files.

Classification	RCV Task	TX Task	Micro-Scheduler
<u>MII Context</u>	MII_RCV (R25 - R29) RCV_DATA (R2 - R9) Length (R18)	TX_CONTEXT (R13 - R17) TX Data (R2 - R9)	None
<u>Descriptor Context</u>	RCV_QUEUE_DESC_REG (R20 - R21)	QUEUE_DESC_REG (R2 - R3)	None
<u>Temp. Reg.</u>	RCV_TEMP_REG_x (R20, R21, R13)	TEMP_REG_x (R0, R4, R2, R3) Others (R10, R11)	TEMP_REG_2 (R4)
<u>Perm. Reg.</u>	None	TX_DATA_POINTER (R1.b3) TX Flags (R22.b3)	TASK_TABLE_ROW0 (R19) CURRENT_TASK_POINTER (R1.b2)

Table 7: PRU Register Usage

3.1.5 Micro Scheduler Task Design

Micro Scheduler is central to our EMAC Design. It schedules various tasks which collectively implement the functionality of EMAC. Simultaneous receive and transmit on both the Ethernet ports at the same time is not possible without Micro-Scheduler. It checks various events and schedules appropriate tasks depending on the outcome of those checks.

Micro-Scheduler schedules the tasks in a round-robin manner. Below, tasks are arranged in the decreasing priority:

1. Receive (RX) Task
2. Transmit (TX) Task
3. Statistics Task

Micro-Scheduler (MS) starts with scheduling the RX Task and then executes the other tasks in the round-robin scheme. It also checks for various events between the execution of two tasks and these events can alter the scheduling. MS checks for below events:

1. Start-of-Frame (SOF)

Start-of-Frame signals a new receive packet at the physical port. MS reads the data from the RX L2 FIFO into PRU registers and checks the 6th bit of the R10 register i.e., RX_SOF status bit. If this bit is clear then MS schedules the next task. If this bit is set then it checks whether 18 bytes have been already been received or not. If 18 bytes have not been received then next task is executed otherwise RCV_FB is called. The number of bytes received is stored in R18[5:0] in RX L2 Bank. Refer MII RT Functional Spec for more details.

2. Receive End-of-Frame (RX_EOF)

Receive End-of-Frame event signals that a frame has been completely received. MS checks this event only if there is an ongoing receive activity at the Port. This event is routed through the R31 register to the two PRU's. If this event has occurred then MS calls the RCV_LB function otherwise it schedules the execution of the next task.

3. Transmit End-of-Frame (TX_EOF)

Transmit End-of-Frame event signals the completion of the transmission of a frame and that TX FIFO is empty. Once this event has occurred, PRU can start the transmission of the next packet. This event is realized through the underflow event on the TX FIFO. When the TX FIFO becomes empty without seeing the TX_EOF command it generates the underflow event. MS checks this event by reading the system event register of the ICSS INTC.

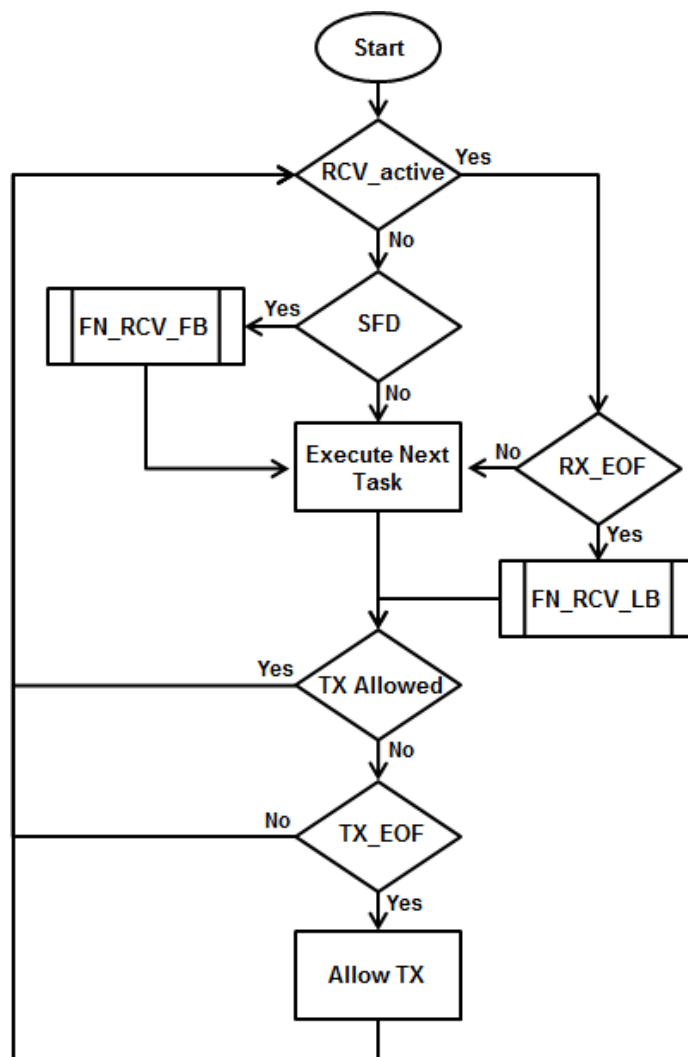


Figure 4: Micro Scheduler Flow Chart

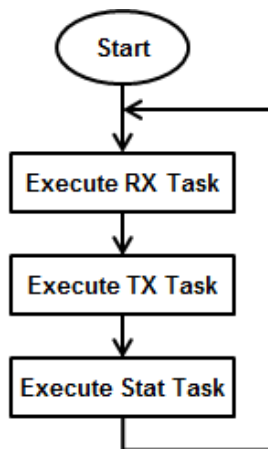


Figure 5: Round-robin approach followed by MS

3.1.6 Receive Task Design

3.1.6.1 Brief Overview of RX L2 FIFO Architecture

The incoming frames are stored in RXL2 FIFO which is composed of 2 banks. Each bank has 32 bytes of data, 16 bytes of status and a 5 bit write pointer. There is one status entry per two bytes. The write pointer gives the info about data entry being written. A status can be volatile or static. A volatile status is one which is not yet complete, and hence cannot be parsed.

Note: The frames are packed contiguously. The buffer does not switch on each EOF.

3.1.6.2 Receive Frame Types

A receive frame can be one of the following types:

1. Unicast Frame
2. Broadcast Frame
3. Multicast Frame

A frame can be classified into one of the above three categories by looking at the destination address (DA) of the frame. If the first bit (LSB) of the first byte of DA is one then it is either multicast or broadcast frame. Further, if the DA is equal to 0xFFFFFFFF then it is broadcast frame otherwise it is multicast frame. If the LSB of the first byte of DA is zero then it is a unicast frame.

Following are actions taken by the Receive Task depending on the type of frame:

1. Unicast Frame

If promiscuous mode is not enabled and If a frame is unicast then its destination address is compared against the interface MAC address of the device/slave. If it matches then the frame is received in one of the host queue depending on the priority of the frame. If it doesn't match then the frame is dropped and dropped frame stats are updated. But if promiscuous mode is enabled then it will accept all unicast packet and forward it to host.

2. Broadcast Frame

Broadcast frame is received in the host queue after checking for broadcast storm i.e., carrying out storm prevention. Storm prevention is only checked if enable by host.

3. Multicast Frame

In EMAC, the handling of multicast frame is the same as that of broadcast frames.

3.1.6.3 Receive Task Design

Receive (RX) Task handles the reception of frames. It is responsible for storing the frames in host queues. If a frame is not being received then Micro-scheduler checks for the start of a receive frame before calling the next task. If Micro-scheduler detects start of receive frame and already 18 bytes have been received then it calls the Receive Task.

RX Task first of all checks whether the port on which frame has arrived is enabled or not. If that port is not enabled then the incoming frame is dropped. RX Task then checks whether the source address in the incoming frame matches with the interface MAC address of the slave. If it matches then the frame is dropped. If an incoming frame is not dropped then RX context is initialized.

Following are the main parameters in the RX context:

1. `host_rcv_flag`: This flag is set when the frame is received for the host port.
2. `qos_queue`: This field stores the priority queue as determined by the frame.
3. `byte_cntr`: Number of bytes already received for the frame being received.
4. `buffer_index`: Offset of the data buffer in L3 RAM where the receive frame is stored.
5. `base_buffer_index`: Offset of the data buffer in L3 RAM corresponding to the first buffer descriptor in the queue.
6. `rcv_queue_pointer`: Offset of the receive queue which is selected for the receive frame.
7. `base_buffer_desc_offset`: Offset of the base buffer descriptor for the queue selected.
8. `top_most_buffer_desc_offset`: Offset of the top most buffer descriptor for the queue selected.

RX context is initialized in the beginning and updated throughout the reception of the frame. When RX Task enters, it reads in the RX context from the scratch pad and saves it back before it exits. Offsets of the top most buffer descriptor is used to quickly determine if there is a wrap around in the receive queue.

RX Task is partitioned into following three parts:

1. Receive First Block (FN_RCV_FB)

This block is executed if there is a new receive frame at the port. First it parses the incoming frame to determine whether it is received or dropped. If a frame is to be received then “`host_rcv_flag`” is set. Depending on the status of the flag it initializes the RX context for host receive or skips initialization if the frame is to be dropped and updates stats. At the end the RX context for host receive is saved. If the firmware is running slow compared to RX L2 FIFO filling up, the control is transferred to FN_RCV_NB otherwise it returns to the Micro-scheduler.

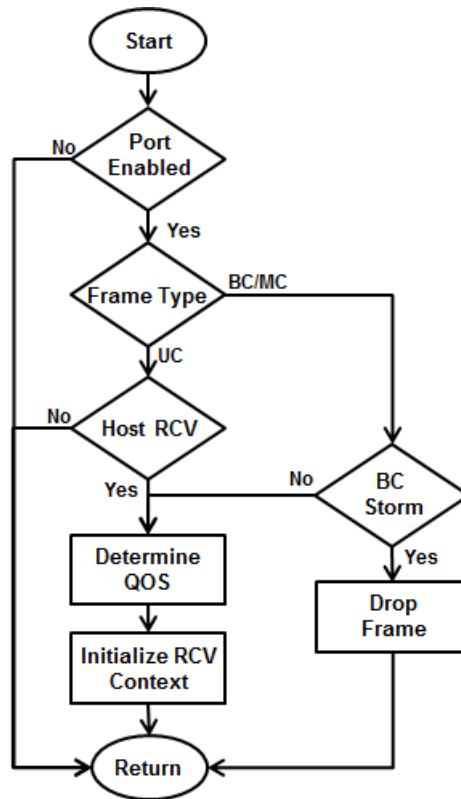


Figure 6: RCV_FB Flow Chart

2. Receive Next Block (FN_RCV_NB)

This block stores the data of the receive frame in the host queue. Maximum it can store a block of 32 bytes per call. Once a queue is acquired, it starts storing the frame in blocks of 32 bytes. Then it checks whether new 32 bytes have become available, if yes then it stores them otherwise transfers the control to Micro-scheduler. After storing the bank index flag, rx_bank_index, is flipped to remember that the next time the data is stored from other bank of RX L2 FIFO.

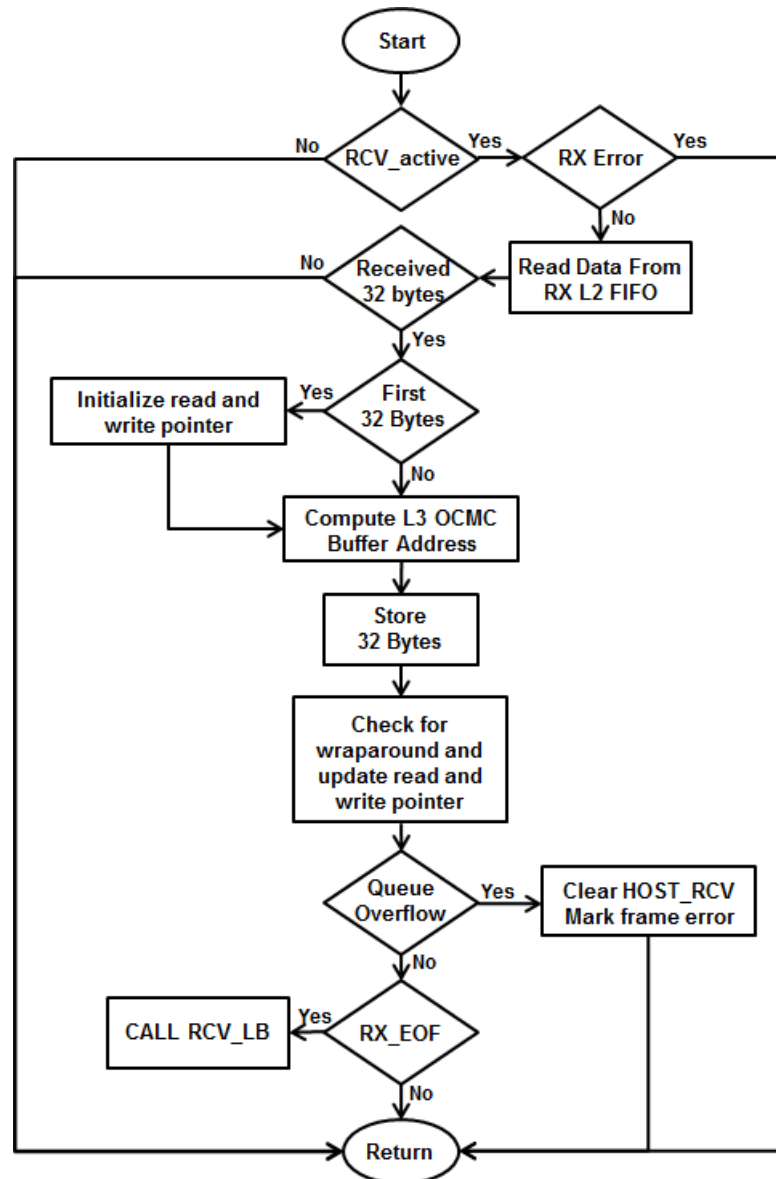


Figure 7: RCV_NB Flow Chart

3. Receive Last Block (FN_RCV_LB)

This block is executed when the RX EOF event has occurred. This block may store less than 32 bytes or exact 32 bytes or more than 32 bytes. In-case where it stores more than 32 bytes of data, it stores from both the banks of RX L2 FIFO. After storing the data of the received frame it updates the first buffer descriptor for the Rx frame with the length of frame and port number on which frame was received. It then updates the queue descriptor to complete the reception of the frame. Only when the queue descriptor is updated the Host comes to know about the received frame. Then it releases the acquired queue and generates an interrupt to host when the frame is received in the host queue. It also sets a flag, RX_STAT_PEND, to signal the Statistics Task. At last it

clears the RCV_active bit, clears the RX_EOF event in ICSS INTC and transfers control back to the Micro-scheduler.

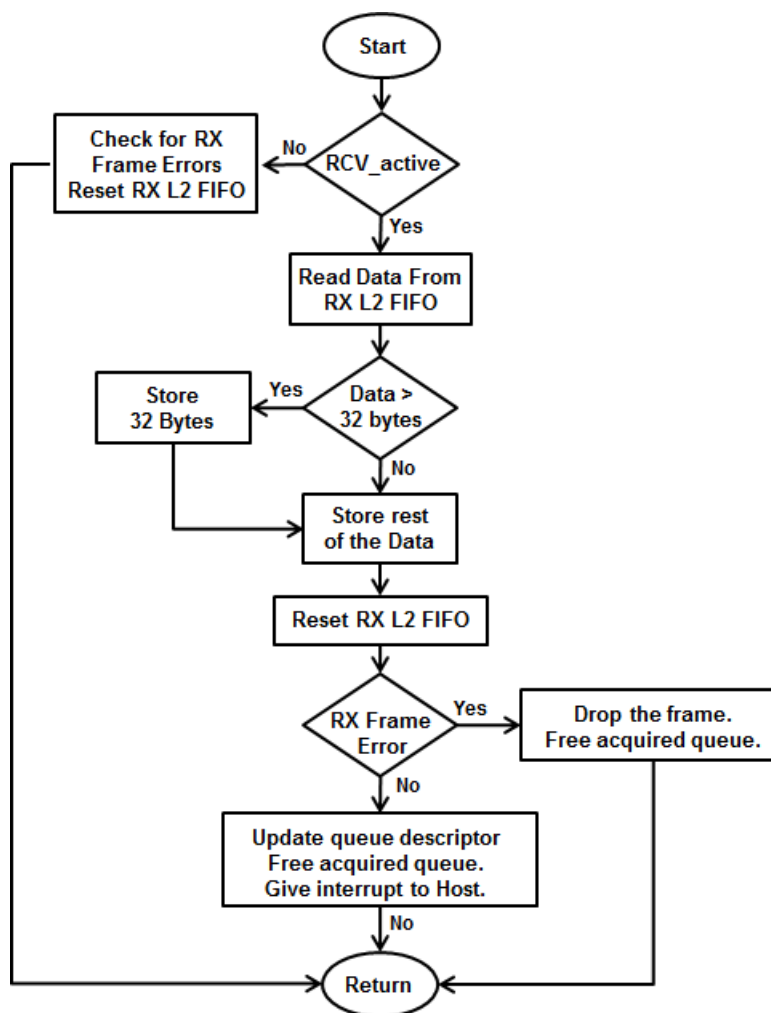


Figure 8: RCV_LB Flow Chart

3.1.7 Buffer Descriptors, Queue Descriptors and Receive Context

The way incoming packets are stored in L3 OCMC RAM has a lot to do with the manner in which they are received from FIFO. Incoming data in the form of 32 byte chunks from the Rx FIFO is stored as is by the Firmware to L3 OCMC RAM. These chunks are stored in contiguous manner. So a 64 byte packet would get stored in two blocks. Each such 32 byte block is in turn pointed to by a buffer descriptor stored in ICSS Shared RAM. Each queue consists of a string of such buffer descriptors kept contiguously on the ICSS Shared RAM. A description of the Buffer Descriptor is given below.

Bit(s)	Name	Meaning
0-7	Index	Points to index in buffer queue, max 256 x 32 byte blocks can be addressed
8-12	Block_length	Number of valid bytes in this specific block. Will be <=32 bytes on last block of packet.
13	More	"More" bit indicating that there are more blocks.
14	Shadow	Indicates that "index" is pointing into shadow buffer. (Not in EMAC)
15	TimeStamp	Indicates that this packet has time stamp in separate buffer - only needed if PTCP runs on host.
16-17	Port	Different meaning for ingress and egress. Ingress: Port=0 indicates PHY Port 1 and Port=1 indicates PHY Port 2. Egress: Port=0 sends on PHY Port 1 and Port=1 sends on PHY Port 2. Port=2 goes over MAC table look-up.
18-28	Length	11 bit of total packet length which is put into first BD only so that host accesses only one BD.
29	VlanTag	Indicates that packet has Length/Type field of 0x8100 with VLAN tag in following byte.
30	Broadcast	Indicates that packet goes out on both physical ports, there will be two BD but only one buffer.
31	Error	Indicates there was an error in the packet.

Table 8: Buffer Descriptor Bits

The first descriptor contains the length of the packet through which the driver knows how many bytes it will have to copy. Since the buffer descriptors are contiguous in memory no additional pointers are required.

Bit(s)	Name	Meaning
0-15	Rd_ptr	Read pointer. This points to the last buffer descriptor that points to valid data, when FW RX Task puts data it increments the Read pointer
16-31	Wr_ptr	Write pointer. This points to the bottom of the first buffer descriptor that contains the data, when RX Task on Driver reads the data it increments this. When read pointer equals write pointer there is no data in the buffers.
32-39	busy_s	Is just a single bit. The busy bit is set by the driver to indicate to the firmware that there is an ongoing copy, firmware does not use the memory during that time.
40-47	status	Queue status.
48-55	max_fill_level	Maximum fill level of the queue. In bytes.
56-63	overflow_cnt	Number of times the queue has overflown

Table 9: Queue Descriptor Bits

Using the queue descriptor both firmware and driver know which is the buffer descriptor which points to the current data and combining this with the data from receive context which tells us

where the top and bottom buffer descriptors are located it's easy to tell if there is a wraparound condition in the queue.

3.1.8 Quality of Service (QoS)

Receive Task implements Quality of Service (QoS) for all the received frames. Firmware uses the VLAN tag to determine the priority of received frame. There are four priority receive queues for the host, two each for receive from each port. Queue priority 0 and queue priority 1 are used for receiving from Port 1 where queue priority 0 is higher priority. Queue priority 2 and queue priority 3 are used for receiving from Port 2 where queue priority 2 is higher priority. A received frame is parsed using quality of service rules to determine in which queue the frame would be received. Following QoS rules are implemented by the firmware:

1. All the non-VLAN tagged frames are stored in the lowest priority queue (Queue Priority 1 for Port 1 and Queue Priority 3 for Port 2).
2. VLAN tagged frames with “Priority Code Point (PCP)” value of 5, 6 and 7 are stored in highest priority queue (Queue Priority 0 for Port 1 and Queue Priority 2 for Port 2).
3. VLAN tagged frames with “Priority code point (PCP)” value of 1, 2, 3 and 4 are stored in low priority queue (Queue Priority 1 for Port 1 and Queue Priority 3 for Port 2).

3.1.9 Transmit Task design

3.1.9.1 Transmit Task design

Transmit task scans the send queues to determine whether there is a frame to be transmitted. It first looks at the highest priority queue and if it is empty then only checks the lower priority queues. If all the send queues are empty then it returns to the scheduler.

Whenever TX task is entered, it first checks XMT_active to determine whether there is an ongoing transmission of a frame and if it is set then TX task fills the next bytes into the TX FIFO. If XMT_active is clear then send queues are looked up to find whether there is a frame to transmit. If there is a frame to be transmitted then it initializes the TX context. Following are the main parameters in the TX context:

1. BUFFER_INDEX: Offset address of the data buffer in L3 RAM which contains first 32 bytes of the frame.
2. Packet_Length: Length of the transmit frame in number of bytes.
3. BYTE_CNT: Number of bytes already pushed to the TX FIFO for the transmit frame.
4. BUFFER_DESC_OFFSET: Offset of the first buffer descriptor for the transmit frame. First buffer descriptor contains the length of transmit frame.
5. BASE_BUFFER_DESC_OFFSET: Offset of the base buffer descriptor of the queue in which transmit frame is queued.
6. BUFFER_OFFSET: Offset of the base data buffer in the L3 RAM for the queue containing frame to be transmitted.
7. TOP_MOST_BUFFER_INDEX: Offset of the top most data buffer in the L3 RAM for the queue containing frame to be transmitted.
8. TOP_MOST_BUFFER_DESC_OFFSET: Offset of the top most buffer descriptor of the queue containing frame to be transmitted.

TX context is initialized in the beginning and updated throughout the transmission of the frame. When TX task enters it reads in the TX context from the scratch pad and saves it back before it exits. Offsets of the top most buffer descriptor and data buffer are used to quickly determine if there is a wrap around in the send queue.

TX Task is partitioned into following three parts:

1. Transmit First Block (XMT_FB)

This block of code first checks in the highest priority queue whether there is a frame to be transmitted. If not, then it checks in second highest priority and so on. Once it finds a pending frame then it initializes the TX context and set's the XMT_active bit. It fetches the first 32 bytes of transmit frame from the L3 RAM and pushes this data in the TX FIFO. After pushing first two bytes into the TX FIFO, it enables transmit of the frame so that frame transmission can start as soon as possible. After pushing first 32 bytes of data, it checks whether RX_EOF event has occurred or not. If this event has not occurred then it fetches next block of data and pushes further 24 bytes into the TX FIFO. Idea is to start with a completely filled TX FIFO so that there is more time for PRU to come back and fill TX FIFO.

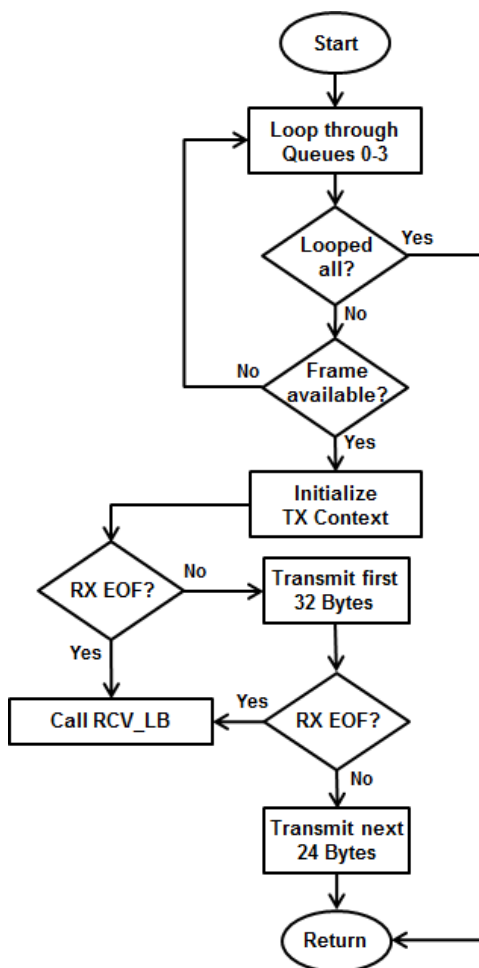


Figure 9: XMT_FB Flow Chart

2. Transmit Next Block (XMT_NB)

This block of code fetches and transmits rest of the frame apart from the last 32 bytes of data. It computes the TX FIFO fill level using the IEP counter (Revision1 device types) or reads it directly from MII RT register (Revision2 device types) and fills the available free space in TX FIFO with the subsequent data of the frame. Maximum it fills 32 bytes of data.

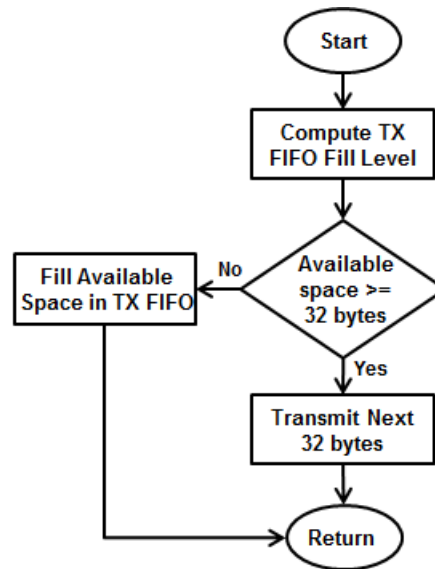


Figure 10: XMT_NB Flow Chart

3. Transmit Last Block (XMT_LB)

This block is executed when there are 32 bytes or less to be transmitted. After pushing the remaining bytes into the TX FIFO, it also pushes the CRC for the frame. It also updates the read pointer in the queue to indicate that frame has been transmitted.

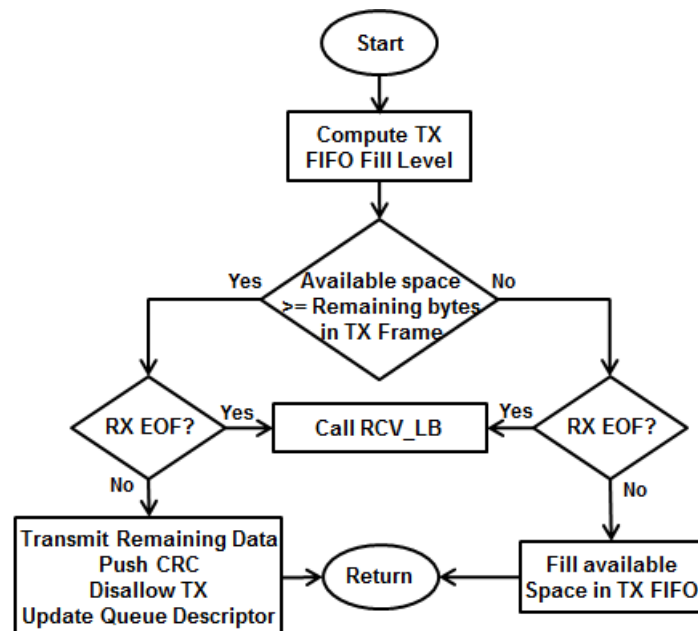


Figure 11: XMT_LB Flow Chart

3.1.10 Statistics Task

Statistics on PRU

The statistics task is called by the scheduler periodically in a round robin manner along with RX and TX task. The stats task checks for two flags RX_STAT_PEND and TX_STAT_PEND which are in turn set by the RX and TX task respectively (these flags are only set for non-error frames). Inside the stat task regular counters like multicast/broadcast/unicast counters along with binning counters are updated. Error cases like CRC count, oversize, undersize frames, RX Error, SFD Error, dropped frames etc. are not part of the stat task. These counters are incremented wherever and wherever the error is detected. In addition to this Half Duplex counters like late collision, excess collision are also not part of the stat task.

Besides these statistics the firmware does binning (segregating based on size of packet) of RX and TX frames. In the new design firmware does most of the statistics and all counters except one on Host are redundant. They are left there to help in debugging. The one counter exclusive to Host is the rxUnknownProtocol.

Statistics on Host

On the Host Core, all statistics related functions are in the file icss_emacStatistics.c. The user has to call ICSS_EmacReadStats() function which reads statistics from the shared memory in PRU and copies them into the corresponding structure in Host.

- pruStat : Contains PRU statistics
- hostStat : Contains Statistics collected by Host (Redundant)

3.1.11 Storm Prevention

Storm prevention is primarily done on PRUs using a credit based scheme. It is explained below.

- The Host writes the number of Multicast+Broadcast packets allowed in a 100ms interval in DRAM of PRU. (STORM_PREVENTION_OFFSET)
- This value (credits) can be configured using the API `setCreditValue()`
- As soon as the PRU encounters a Multicast/Broadcast packet it decrements the value written in memory by 1 and allows the packet to pass through. If the value goes to 0 the packet is dropped
- At the end of every 100ms interval the host writes the value once again. Function `ICSS_EmacResetStormPreventionCounter()` in file `icss_StormControl.c`
- Storm Control can be enabled and disabled on a per port basis using `ICSS_EmacEnableStormPrevention` and `ICSS_EmacDisableStormPrevention()`. It can be initialized using `ICSS_EmacInitStormPreventionTable()`.
- The default credit value is defined as `DEFAULT_CREDITS` in `icss_emacStormControl.h` and is equal to 2000.
- APIs defined in `icss_emacStormControl.c`.

3.1.12 Half Duplex Support

The latest firmware supports half duplex and associated statistics like late collision, multiple collisions etc. The exception being that unlike the recommendation, frames are not dropped after the maximum threshold for multiple collisions is reached, only a note is made of it. Half Duplex requires proper pin-muxing to enable collision and carrier sense lines from the PHY. This is dependent on the board type hence enable the external variable `halfDuplexEnable` which is part of EMAC configuration to enable half duplex in EMAC. If this configuration is enabled, the EMAC driver assumes that proper pin-muxing has been done.

3.1.13 Link Status Change detection

Link change is monitored via an Interrupt on HOST. When there is a change in link state such as speed/duplexity or a link up/down event, an interrupt is triggered from the MDIO which invokes the ISR `link0ISR/link1ISR` on HOST for Port0/Port1 respectively. The ISR modifies the link state information in two variables.

- `PORT_STATUS_OFFSET` – Contains link up/down and Half Duplex/Full Duplex information.
- `PHY_SPEED_OFFSET` – Contains PHY speed information.

3.1.14 EMAC Time Triggered Send

3.1.14.1 Brief Overview of TTS

The EMAC time triggered send is used to expand classical Ethernet to meet deterministic, time-critical or safety-relevant conditions. TTS has been designed by using the IEP Counter and Compare Registers. Queue 0 is reserved as the real-time queue. All packets in queue 0 are cyclic packets. On the other hand, packets from other queues are acyclic packets. Cyclic packets are sent at triggered instances, as programmed in compare registers, whereas acyclic packets are sent based on time availability, as shown in subsequent figures throughout this section.

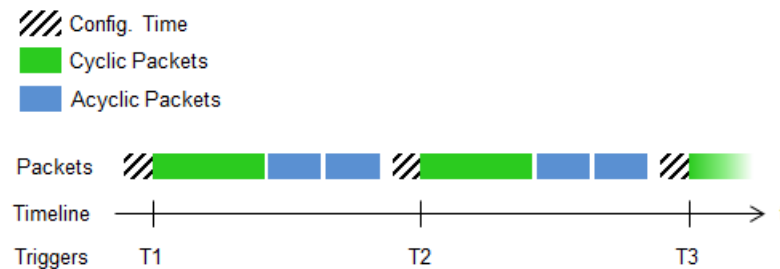


Figure 12: Time Triggered Send Overview

3.1.14.2 TTS Register Usage and Source Files

Time triggered send (TTS) has been implemented as part of the TX Task. The code is spread across two files, `emac_MII_Xmt.asm` and `micro_scheduler.asm`. Different files involved in TTS implementation are documented below.

File Name	Description
<code>emac_MII_Xmt.asm</code>	Minimal changes in existing Tx Task. Used to incorporate calls for different TTS functions and macros.
<code>micro_scheduler.asm</code>	Minimal changes. Used to incorporate calls for different TTS functions and macros.
<code>emac_tts.asm</code>	Contains TTS functions.
<code>emac_tts.h</code>	Contains TTS macros and pre-processor directives.
<code>icss_emacSwitch.h</code>	Contains TTS offsets. These offsets are also used by host to enable TTS.

Table 10: TTS Source Code Files List

No registers are reserved for TTS usage, except for some bits in the R22 persistent register. The persistent bits reserved for TTS in R22 are documented below.

Bit	Name	Usage
R22.t21	TTS_ENABLE	Set when TTS is enabled by host. If set, all TTS related code will execute as part of Tx Task. If clear, no TTS related code will execute.
R22.t20	TTS_FIRST_SETUP	Set after the first time setup of IEP compare registers is done and TX_EN_MODE is set, in function <code>FN_TTS_IEP_CFG_PRE</code> . Cleared when TTS is disabled.

R22.t19	TTS_RT_QUEUE_PKT	Set when the packet is coming from reserved real-time queue (queue 0). Cleared when packet is coming from non-RT queue (queue 1,2,3).
R22.t18	TTS_FIRST_PKT_done	Set when at least one packet has been transmitted in current cycle. Cleared after every cycle in function FN_TTS_IEP_CFG_PRE.
R22.t17	TTS_FIRST_PKT_since_enable	Set when the first packet has been transmitted after TTS is enabled. Cleared when TTS is disabled. Used for debug purposes such as calculating cycles missed by cyclic packets.
R22.t16	TTS_CMP_3456_FIRST_Exception	This is specific to Revision1 firmware. Since Revision1 device types have a 32-bit counter, we need to handle the counter wraparound properly. The first time triggers are programmed into compare registers, the compare registers are not enabled if the triggers are supposed to occur in next cycle of IEP counter. They are enabled after wraparound. This is because if triggers are enabled immediately, the IEP_Counter>Compare Value and Compare Event Enable -> Toggle events become true, and the compare event is hit immediately which is erroneous behavior. Thus this bit is used to keep this in check.
R22.t15	TTS_CMP5_CMP6_in_next_IEP_cycle	This is specific to Revision1 firmware. This bit is used to detect if the compare event being programmed is in current IEP cycle or next IEP cycle. It helps in eliminating boundary erroneous conditions.

Table 11: TTS R22 Bits Usage

NOTE: The queue naming convention used in this documentation for EMAC TTS is as per the EMAC Driver API, i.e., queue 0/1/2/3. The firmware uses a different naming convention, i.e., queue 1/2/3/4. The reserved real-time queue for cyclic packets is the high priority queue 0 (as per Driver API) or queue 1 (as per firmware).

3.1.14.3 TTS Parameters and Memory Map

Below is a list of parameters used by TTS in firmware. These are defined in icss_emacSwitch.h.

Parameter	Size (bytes)	Description
ICSS_EMAC_TTS_BASE_OFFSET	NA	Base start address of TTS offsets. This should always be kept equal to start of EMAC specific DRAM.
ICSS_EMAC_TTS_CYCLE_START_OFFSET	8	The IEP counter value (ns) to be programmed for the first trigger. Provided by the host using the ICSS EMAC IOCTL.
ICSS_EMAC_TTS_CYCLE_PERIOD_OFFSET	4	The cycle period (ns) of triggers for cyclic frames. Provided by the host using the ICSS EMAC IOCTL.
ICSS_EMAC_TTS_CFG_TIME_OFFSET	4	The amount of time available for configuring the next TTS cycle. For most applications this time will also include the time required to receive input, process input, and then generate output (cyclic frames); all of which is normally done by host. Provided by the host using the ICSS EMAC IOCTL.
ICSS_EMAC_TTS_STATUS_OFFSET	4	Stores TTS firmware status. This can be queried by Host at any time using ICSS EMAC IOCTL.
ICSS_EMAC_TTS_MISSED_CYCLE_CNT_OFFSET	4	Stores the number of cycles missed by cyclic frames. This can be queried by Host at any time using ICSS EMAC IOCTL.
ICSS_EMAC_TTS_PREV_TX_SOF	8	The Transmit Start-of-Frame for previous packet. This can be queried by Host at any time using ICSS EMAC IOCTL.
ICSS_EMAC_TTS_CYC_TX_SOF	8	The Transmit Start-of-Frame for previous cyclic packet. This can be queried by Host at any time using ICSS EMAC IOCTL.

Table 12: TTS Control Variables

Memory map for TTS is as follows:

Definition	Address Map	Remarks
TTS Control Variables	0x1E98 - 0xIEC0	Stores TTS Control Variables. 40 bytes starting from EMAC_SPECIFIC_DRAM_START_OFFSET. Present on both, DRAM0 and DRAM1.

Table 13: TTS Memory Map

The bitwise summary for TTS status stored at ICSS_EMAC_TTS_STATUS_OFFSET is as follows:

Bits	Field Name	Description
31:6	RESERVED	Reserved for future use.
5	ICSS_EMAC_TTS_CMP0_CMP7_SETUP	0: CMP0/CMP7 setup not completed. 1: CMP0/CMP7 setup completed. Revision1 specific. Cleared in driver API when TTS is enabled.
4	ICSS_EMAC_TTS_CYC_INTERRUPT_ENABLE	0: Disable interrupt notification for cyclic packet insertion. 1: Enable interrupt notification for cyclic packet insertion. Cleared in driver API when TTS is enabled.
3	ICSS_EMAC_TTS_CYC_TX_SOF_ENABLE	0: Disable storing of cyclic packets' transmit start-of-frame. 1: Enable storing of cyclic packets' transmit start-of-frame. Cleared in driver API when TTS is disabled. Cleared in driver API when TTS is enabled.
2	ICSS_EMAC_TTS_INSERT_CYC_FRAME_EVENT	0: Not in config period. No need to insert cyclic frame at this time. 1: Config period has started. Need to insert cyclic frame. Cleared on every CMP3/CMP4 event and on receiving a cyclic frame in queue 0. Cleared in driver API when TTS is enabled.
1	ICSS_EMAC_TTS_MISSED_CYCLE	0: No cycles missed by cyclic packets. 1: At least one cycle missed by cyclic packets. No. of cycles missed can be checked from ICSS_EMAC_TTS_MISSED_CYCLE_CNT_OFFSET Cleared in driver API when TTS is enabled.
0	ICSS_EMAC_TTS_PRU_ENABLE	0: Disable TTS on PRU. 1: Enable TTS on PRU. Cleared in driver API when TTS is disabled. Cleared in driver API when TTS is enabled.

Table 14: TTS Status Bits

3.1.14.4 TTS Macros

Multiple Macros have been used to implement various TTS related code. Macros have been used to keep the code clean which would otherwise have cluttered all code files especially considering that Revision1 and Revision2 device types have slightly different TTS implementations. The details regarding TTS Macros can be found in Firmware Macros Description section.

3.1.14.5 TTS Functions

The following is the list of TTS related functions and their purpose:

Function	Description
FN_TTS_IEP_CFG_PRE_ICSS_REV2	Handles pre-configuration of TTS cycles in Config Period before every TTS cycle. Pre-configuration involves setting up the compare registers for upcoming cycle and also calculating and storing the Cycle Start Time for next cycle. This is specific to Revision2 because IEP handling is different for Revision1 and Revision2 device types in TTS algorithm.
FN_TTS_IEP_CFG_PRE_ICSS_REV1	Handles pre-configuration of TTS cycles in Config Period before every TTS cycle. Pre-configuration involves setting up the compare registers for upcoming cycle and also calculating and storing the Cycle Start Time for next cycle. This is specific to Revision1 because IEP handling is different for Revision1 and Revision2 device types in TTS algorithm.
FN_TTS_IEP_CFG_CLEAR	Handles clearing of R22 TTS bits and TTS control variables when TTS is disabled. It also disables all compare events.
FN_TTS_PKT_SIZE_CHECK_ICSS_REV2	Handles checking of packet size before the packet can be transmitted to ensure that the pre-configuration of next cycle starts on time. It ensures that the packet can be transmitted within the current cycle before CMP5/CMP6 event. This is specific to Revision2 because IEP handling is different for Revision1 and Revision2 device types in TTS algorithm.
FN_TTS_PKT_SIZE_CHECK_ICSS_REV1	Handles checking of packet size before the packet can be transmitted to ensure that the pre-configuration of next cycle starts on time. It ensures that the packet can be transmitted within the current cycle before CMP5/CMP6 event. This is specific to Revision1 because IEP handling is different for Revision1 and Revision2 device types in TTS algorithm.
FN_TTS_IEP_CMPCFG_ARBITRATION	This is used for arbitration between PRU0 and PRU1 when TTS is enabled on both PRUs and they compete to access the IEP Compare Configuration Register, which is a shared resource.
FN_TTS_EXIT_IEP_CMPCFG_ARBITRATION	This is used for arbitration between PRU0 and PRU1 when TTS is enabled on both PRUs and they compete to access the IEP Compare Configuration Register, which is a shared resource.

Table 15: TTS Functions

3.1.14.6 TTS Debug Provisions

For the cyclic packet to be transmitted exactly at the pre-defined trigger instant, the packet must be present in the high priority queue (queue 0) before the trigger instant. The firmware informs the Host about this by giving an interrupt (if ICSS_EMAC_TTS_CYC_INTERRUPT_ENABLE is set) or by setting ICSS_EMAC_TTS_INSERT_CYC_FRAME_EVENT. TTS provides a way to check if the firmware found any cyclic packet or not. If no cyclic packet is found in the queue, the ICSS_EMAC_TTS_MISSED_CYCLE bit is set in the TTS status. Also, the counter for missed cycles, ICSS_EMAC_TTS_MISSED_CYCLE_CNT_OFFSET, is updated (incremented by 1).

The firmware also provides a way to verify when the cyclic packet was transmitted by saving the transmit start-of-frame. This is done if ICSS_EMAC_TTS_CYC_TX_SOF_ENABLE is set by host.

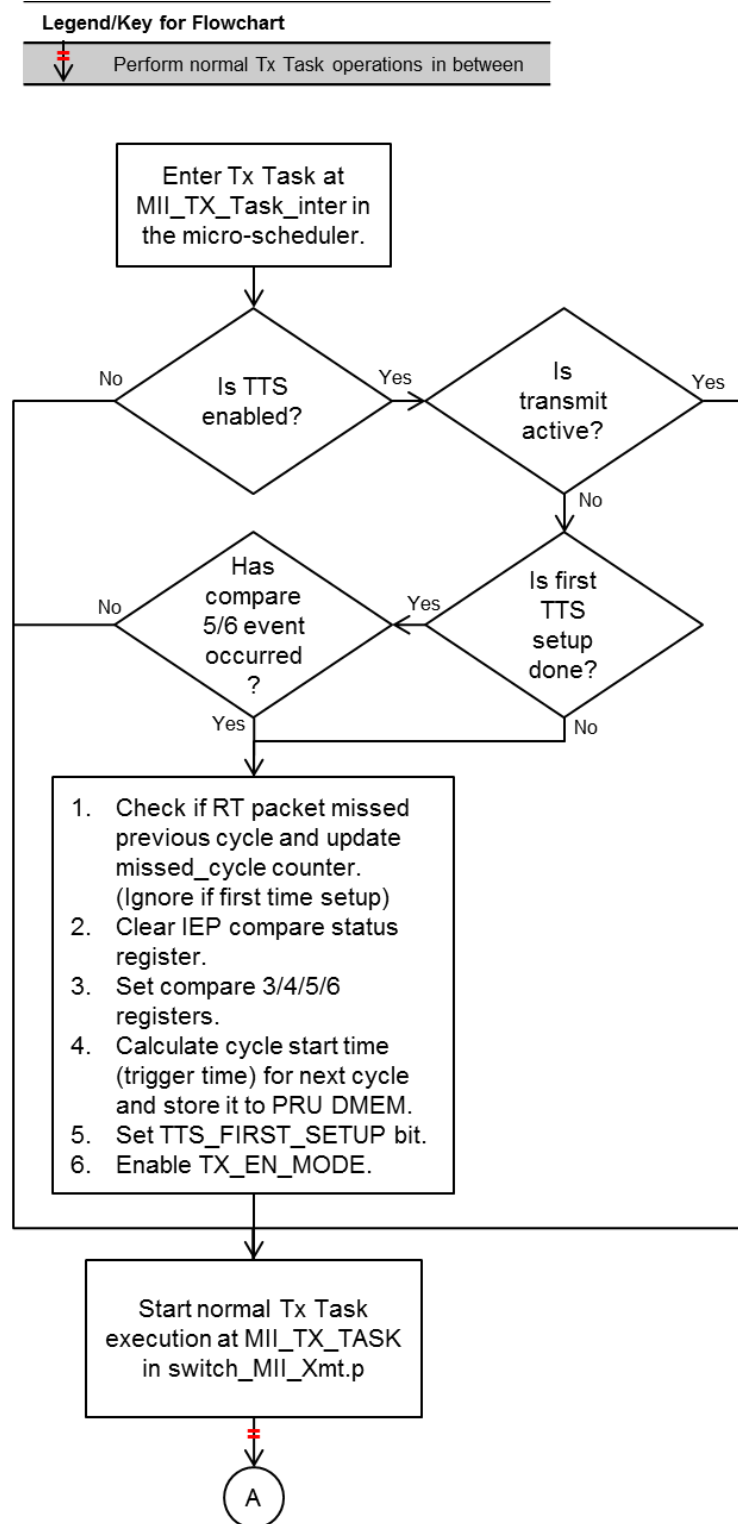
3.1.14.7 TTS Assumptions

The following are the assumptions for TTS implementation:

1. The cycle period is long enough to be able to transmit any cyclic packet in queue 0. Failure to ensure this will cause the cyclic packet with size greater than the allowed size (as per the period), and any cyclic packets following it, to remain in the queue.
2. The configuration time provided (using ICSS EMAC IOCTL) is sufficient for the PRU to configure the next cycle.
3. If the host keeps queueing packets irrespective of the fact whether the packet has been transmitted by the firmware (in accordance with the preset triggered intervals), the packets might get dropped at the driver level if the queue is full. It is assumed that this is taken care of at the host level.

3.1.14.8 Detailed Explanation of TTS

To help understand EMAC TTS better, a detailed flow chart has been provided below. This is just a basic flowchart. The actual implementation might differ considering the fact that Revision1 and Revision2 device types have different IEP counter types.



Legend/Key for Flowchart

 Perform normal Tx Task operations in between

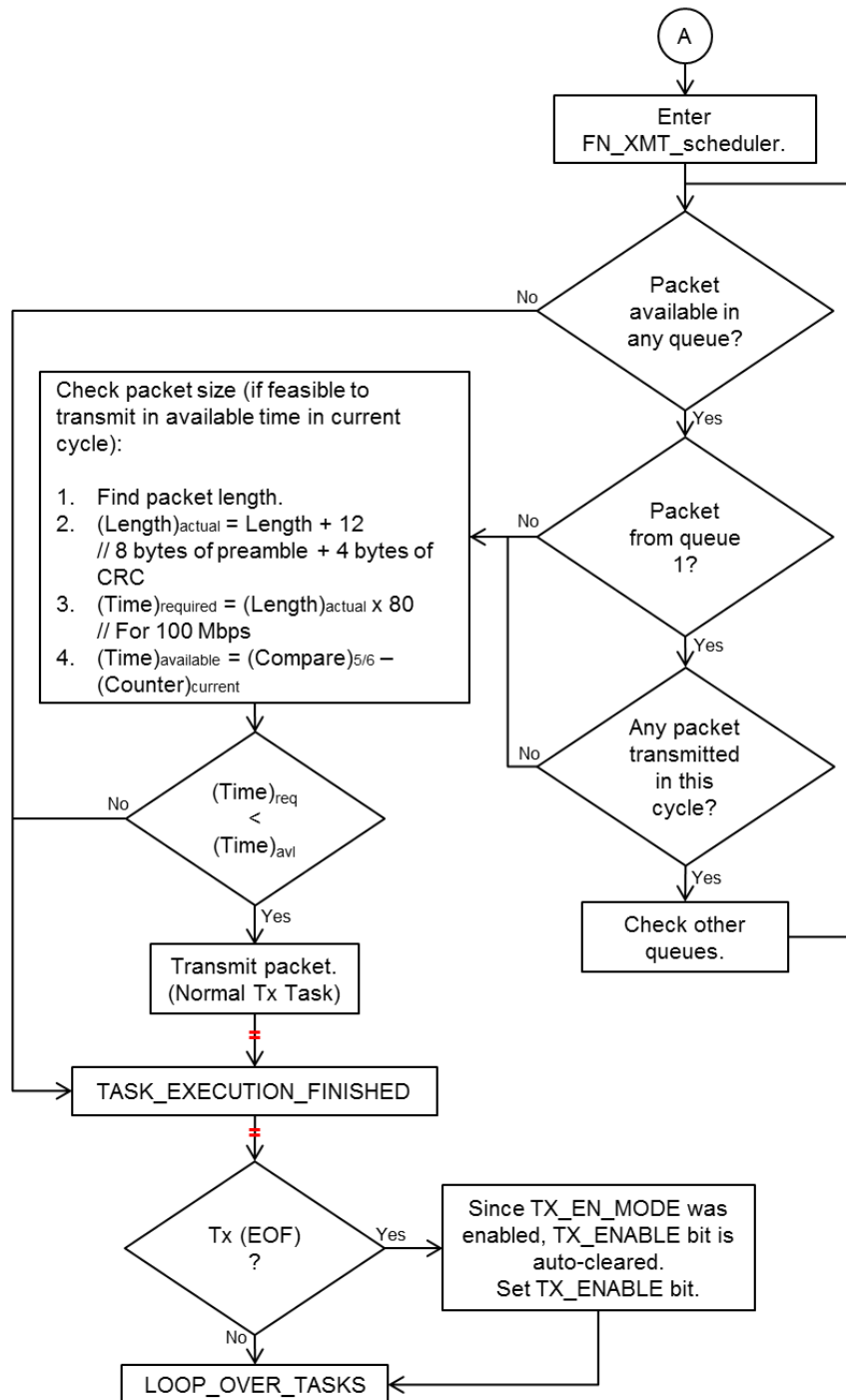


Figure 14: TTS Flow Chart (Part 2)

With TTS enabled, the TX Task first checks if transmit is active. This check is performed because TTS can be enabled/disabled dynamically by the host. If transmit is inactive or once transmit becomes inactive, TX Task checks if TTS has been setup/configured for the first time. If not, we

setup the compare registers, enable TX_EN_MODE in MII_RT_TXCFG register. TX_EN_MODE allows us to use IEP_CMP[3] and IEP_CMP[4] as triggers for TX0 and TX1 respectively, i.e. transmission will start once these events are triggered.

Note: When TX_EN_MODE is enabled, IEP_CMP[3] must be set before the TX will start for TX0, and IEP_CMP[4] for TX1. This is just an additional dependency, to start transmission, to the current set of dependencies, which are FIFO NOT empty and TX_START_DELAY met.

The compare registers usage is as mentioned in the following table:

Compare Register	Purpose
Compare 0 Register	Used to set IEP wraparound to 1 second in Revision1 device types. Also used to detect IEP wraparound for PRU0. Revision1 specific.
Compare 3 Register	Used to set send trigger for TX0 i.e., PRU0
Compare 4 Register	Used to set send trigger for TX1 i.e., PRU1
Compare 5 Register	Used to set configuration trigger for PRU0
Compare 6 Register	Used to set configuration trigger for PRU1
Compare 7 Register	Used to detect IEP wraparound for PRU1. Revision1 specific.

Table 16: TTS Compare Register Usage

After the first iteration, configuration (updating compare registers for next cycle) will only be done if compare 5 event (for PRU0) or compare 6 event (for PRU1) has occurred. These events are the start of configuration time.

Here onwards, TX Task runs and the flow reaches FN_XMT_scheduler. In the transmit scheduler, an additional check has been added to the existing loop for checking packets in different queues. Here, we check if the packet was found in queue 0 (designated high priority queue for cyclic packets) or any other queue. If a cyclic packet is available, we check if a packet has previously been transmitted in the current cycle. This is because of the following reasons:

1. We don't want multiple cyclic packets to be transmitted in the same cycle.
2. We also don't want to transmit a cyclic packet if an acyclic packet has already been transmitted in the current cycle (refer the diagram below).
3. We also want to minimize jitter. This is taken care of by timely notifying Host about when to insert cyclic packet.

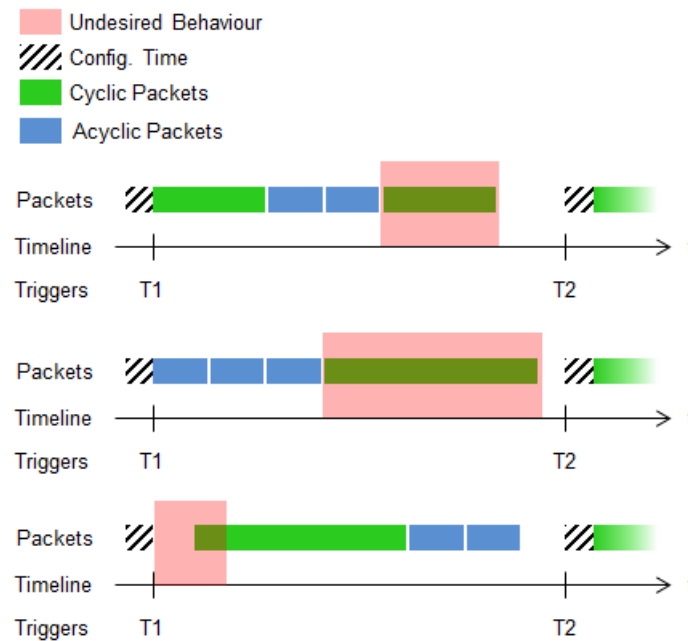


Figure 15: Incorrect Cyclic Packet Transmission in TTS

If a packet has been transmitted, it means the cyclic packet was not added to the queue on time and has missed the cycle or perhaps the cyclic packet under discussion is for next cycle. Refer to EMAC TTS Debug Provisions for more details. In this situation, we skip the cyclic packet and see if any acyclic packet is available. If no packet has been transmitted, it means this is the start of the cycle and the cyclic packet can be sent.

Once the above checks have been performed, we check if there is enough time available in the cycle for the chosen packet to be transmitted. This is done to ensure that the configuration of the next cycle starts on time and the next cyclic packet is sent out on time. Refer the diagram below for more details. If feasible, the packet is transmitted, i.e. put into the TX FIFO. The time check is performed as per the following equations:

1. Find packet length.
2. $(\text{Length})_{\text{actual}} = \text{Length} + 12$ // 8 bytes of preamble + 4 bytes of CRC
3. $(\text{Time})_{\text{required}} = (\text{Length})_{\text{actual}} \times 80$ // For 100 Mbps
4. $(\text{Time})_{\text{available}} = (\text{Compare})_{5/6} - (\text{Counter})_{\text{current}}$
5. if $(\text{Time})_{\text{required}} < (\text{Time})_{\text{available}} \rightarrow$ Transmit packet.

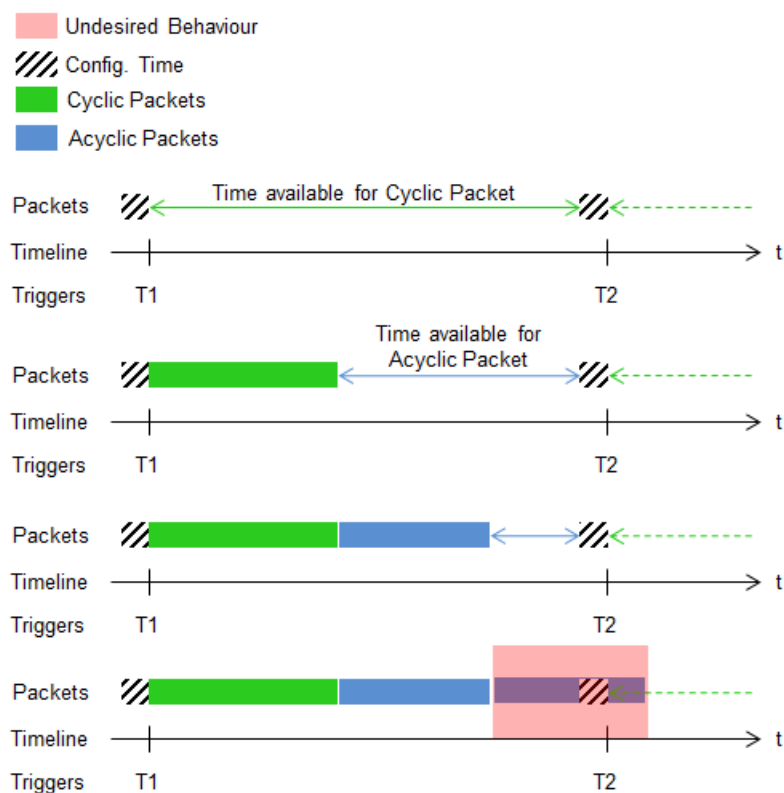


Figure 16: Time Availability Check in TTS

In the end, once the task execution is finished and TX EOF is reached, we need to set TX_ENABLE bit in MII_RT_TXCFG register. This is because when TX_EN_MODE is enabled, TX_ENABLE is auto-cleared.

3.2 Firmware Macros Description

ICSS EMAC Macros are as follows:

Macro	Source File	Remarks
M_MS_TEF_ICSS_REV1	micro_scheduler.h	Set of instructions to execute when Task Execution is Finished (TEF) for Revision1 device types
M_MS_TEF_ICSS_REV2	micro_scheduler.h	Set of instructions to execute when Task Execution is Finished (TEF) for Revision2 device types
M_MS_RX_EOF_CHECK_ICSS_REV1	micro_scheduler.h	Check for RX EOF on Revision1 device types
M_MS_RX_EOF_CHECK_ICSS_REV2	micro_scheduler.h	Check for RX EOF on Revision2 device types
M_RCV_RX_EOF_CHECK_ICSS_REV1	emac_MII_Rcv.h	Check for RX EOF on Revision1 device types
M_RCV_RX_EOF_CHECK_ICSS_REV2	emac_MII_Rcv.h	Check for RX EOF on Revision2 device types
M_RCV_RX_EOF_CLEAR_INTC_ICSS_REV1	emac_MII_Rcv.h	Clear RX EOF system event in INTC on Revision1 device types
M_RCV_RX_EOF_CLEAR_INTC_ICSS_REV2	emac_MII_Rcv.h	Clear RX EOF system event in INTC on Revision2 device types
M_XMT_RX_EOF_CHECK_ICSS_REV1	emac_MII_Xmt.h	Check for RX EOF on Revision1 device types
M_XMT_RX_EOF_CHECK_ICSS_REV2	emac_MII_Xmt.h	Check for RX EOF on Revision2 device types
M_XMT_GET_TXSOF_ICSS_REV1	emac_MII_Xmt.h	Get TX SOF on Revision1 device types for fill level calculation
M_XMT_FILL_LEVEL_CALC_ICSS_REV1	emac_MII_Xmt.h	Fill level calculation on Revision1 device types
M_XMT_FILL_LEVEL_CALC_ICSS_REV2	emac_MII_Xmt.h	Fill level calculation on Revision2 device types
M_XMT_INSERT_CRC_ICSS_REV1	emac_MII_Xmt.h	Insert the CRC in outgoing frame on SOC ICSS_REV1
M_XMT_INSERT_CRC_ICSS_REV2	emac_MII_Xmt.h	Insert the CRC in outgoing frame on SOC ICSS_REV2
M_XMT_UNDER_OVERFLOW_CHECK_ICSS_REV2	emac_MII_Xmt.h	Check underflow and overflow flags on Revision2 device types
M_TTS_XMT_SCHEDULER	emac_tts.h	Checks which queue has the packet and sets appropriate bits (scheduler modification for TTS)
M_TTS_MISSED_CYCLE_CHECK	emac_tts.h	Checks if cycle was missed by RT frame and updates missed_cycle counter
M_CHECK_TTS_ENABLE	emac_tts.h	Check if time triggered send has been enabled by the host
M_TTS_SET_CMP0_IEP_WRAP	emac_tts.h	Sets Compare 0 IEP to wraparound at 1s. (Revision1 specific)
M_TTS_CFG_CONDITIONAL_CONSTRUCT	emac_tts.h	Check essential conditions before TTS configuration
M_SET_MIIRT_TXCFG_TX_ENABLE	emac_tts.h	Set TX_ENABLE bit in MIIRT_TXCFG register if TTS is enabled
M_TTS_CMP0_CMP7_CHECK	emac_tts.h	Checks for CMP0 and CMP7 event, which are set for IEP counter wraparound (Revision1 specific)
M_TTS_CMP3_CMP4_CHECK	emac_tts.h	Checks for CMP3 and CMP4 event and then clears ICSS_EMAC_TTS_INSERT_CYC

		FRAME_EVENT
M_TTS_CYC_FRAME_NOTIFICATION	emac_tts.h	Sets ICSS_EMAC_TTS_INSERT_CYC_FRAME_EVENT bit to notify that it is time to insert cyc frame
M_TTS_TX_SOF_PREV_STORE	emac_tts.h	Saves TX_SOF of previous packet to PRU DMEM0/1
M_TTS_TX_SOF_CYC_STORE	emac_tts.h	Saves TX_SOF of current cyclic packet
M_TTS_TX_SOF_COMPARE_ICSS_REV_2	emac_tts.h	Compares previous and current TX SOF
M_TTS_TX_SOF_COMPARE_ICSS_REV_1	emac_tts.h	Compares previous and current TX SOF
M_TTS_FIFO_FILL_MOD	emac_tts.h	Needed to modify FIFO fill level based on CMP event status

Table 17: ICSS EMAC Macros

3.3 Firmware Sources Description

ICSS EMAC firmware source code includes the following files:

Source File	Remarks
firmware_version.h	ICSS EMAC Firmware Version Control
icss_defines.h	ICSS Global Defines
emac_tts.h	ICSS EMAC Time Triggered Send Macros and Defines
emac_tts.asm	ICSS EMAC Time Triggered Send Functions
icss_iep_reg.h	ICSS Industrial Ethernet Peripheral Registers Definition
icss_intc_reg.h	ICSS Interrupt Controller Module Registers Definition
icss_macros.h	Implements Common Macros & Defines
icss_miirt_regs.h	ICSS MII_RT Module Registers Definition
icss_emacSwitch.h	Definitions and Mapping of Ethernet MAC over PRU
micro_scheduler.h	ICSS Defines and Macros used by Micro_Scheduler
micro_scheduler.asm	Round-robin based Micro_Scheduler which controls program flow
emac_MII_Rcv.h	Defines and Macros to be used in Receive Task
emac_MII_Rcv.asm	Receive Task Functions
emac_MII_Xmt.h	Defines and Macros to be used in Transmit Task
emac_MII_Xmt.asm	Transmit Task Functions
emac_statistics.asm	Statistics Task

Table 18: Firmware Sources Description

4 Revision History

Version #	Date	Author Name	Revision History
0.1	14 June 2016	Anjandeeep Sahni	First Draft Based on AM335x_ICSS_Switch_Firmware_Design.pdf
0.2	07 July 2016	Anjandeeep Sahni	Fixed typo in QoS section.
0.3	28 June 2017	Suraj Das	Modified the source code & design guide to mention PRU revision. Modified files to used with newer file format i.e. *.asm, *.h. Also, mentioned modified function based on new code.
0.4	6 July 2017	Suraj Das	Removed MII spec table as per review comments
0.5	6 September 2017	Suraj Das	Updated with promiscuous mode information

Table 19: Revision History